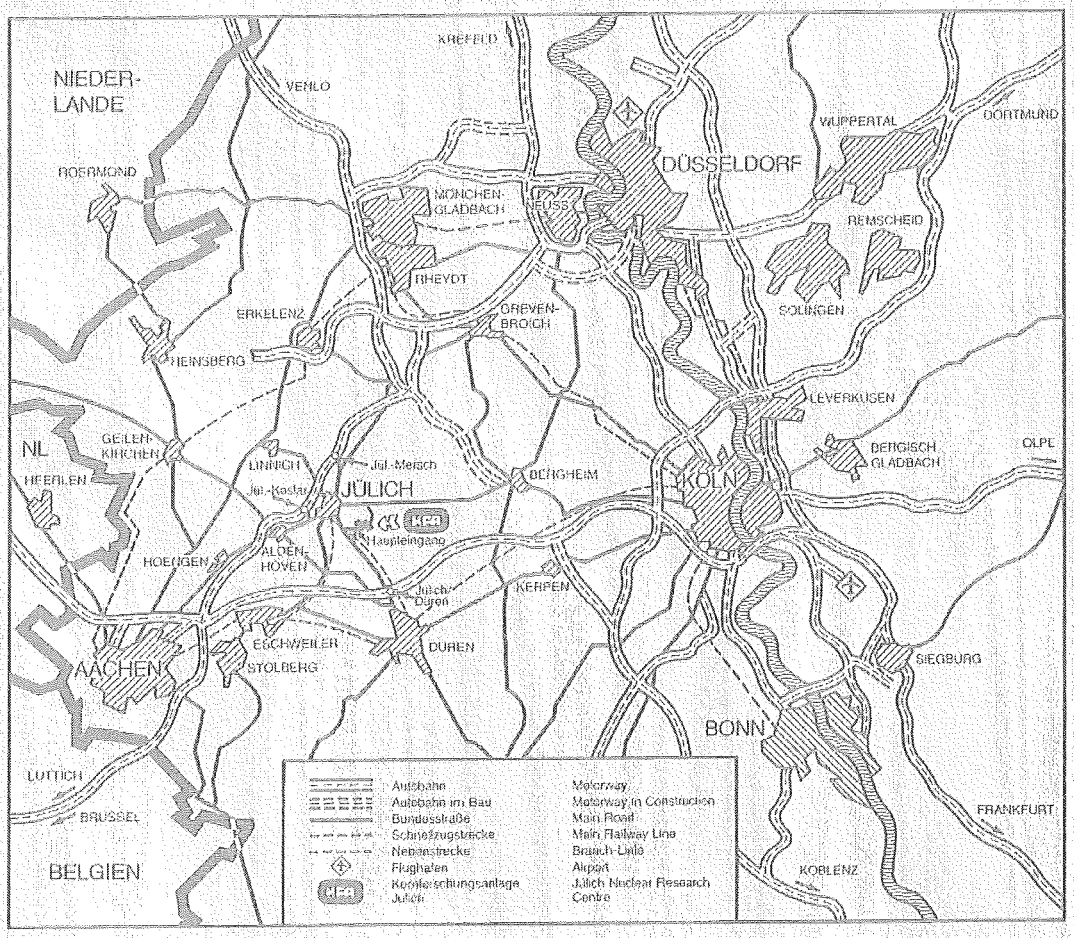
KERNFORSCHUNGSANLAGE JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik

Graphenalgorithmen für MIMD-Rechner

von

Renate Gurke

Jül - Spez - 355
Juni 1986
ISSN 0343-7639



Als Manuskript gedruckt

Spezielle Berichte der Kernforschungsanlage Jülich – Nr. 355

Zentralinstitut für Angewandte Mathematik Jül - Spez - 355

Zu beziehen durch: ZENTRALBIBLIOTHEK der Kernforschungsanlage Jülich GmbH

Postfach 1913 · D-5170 Jülich (Bundesrepublik Deutschland)

Telefon: 02461/610 · Telex: 833556-0 kf d

Graphenalgorithmen für MIMD-Rechner

von

Renate Gurke

Diese Arbeit wurde als Diplomarbeit am Institut für Allgemeine Elektrotechnik und Datenverarbeitungssysteme der Rheinisch-Westfälischen Technischen Hochschule Aachen bei Herrn Prof. Dr.-Ing. O. Lange im Zentralinstitut für Angewandte Mathematik der KFA Jülich angefertigt.

Für die Unterstützung durch das Institut bedanke ich mich beim Institutsdirektor Herrn Dr. F. Hoßfeld.

1.0 EINLEITUNG

Aufgrund der raschen Entwicklung der Halbleitertechnologie und der damit verbundenen Senkung der Hardware-Kosten konnten in der jüngsten Vergangenheit ständig steigende Rechnerleistungen bei vertretbaren Kosten bereitgestellt werden. Doch mit der Verfügbarkeit größerer Rechnerleistungen steigerte sich gleichzeitig auch das Verlangen, komplexere Aufgabenstellungen lösen zu können. Mit einer Verkürzung der Schaltzeiten allein konnte der Wunsch nicht erfüllt werden. Daher wurden Parallelprozessor-Architekturen konzipiert und gewannen an Bedeutung. Mit Hilfe neuer Hardware wurden Rechnerkonzepte entwickelt, die den Parallelismus in Anwenderprogrammen auf vier verschiedenen Ebenen ausnutzen können: durch parallele Ausführung einzelner Operationen, durch Schleifenvektorisierung, durch parallele Ausführung von Unterprogrammen sowie durch parallele Ausführung von mehreren Benutzerprogrammen.

Erste kommerzielle Erfolge sind durch Pipeline-Rechner, d.h. SIMD-Rechner erzielt worden. In den letzten Jahren ist jedoch auch die Anzahl der Multiprozessorsysteme auf dem Markt gestiegen. Ähnlich verlief auch die Entwicklung entsprechender Algorithmen. Sowohl für numerische als auch für nichtnumerische Probleme existiert bisher eine große Zahl von SIMD-Algorithmen, die oft mit geringem Aufwand aus den sequentiellen Algorithmen hergeleitet werden können. Die meisten Implementationen von Algorithmen auf MIMD-Rechnern gehören in den Bereich der Linearen Algebra zur Lösung naturwissenschaftlich-technischer Probleme. Aber auch die Bedeutung nichtnumerischer Algorithmen hat sich erhöht. Da Anwendungen der Graphentheorie in Technik und Wissenschaft weit verbreitet sind und realistische Graphenprobleme in der Praxis viel Rechenzeit benötigen, wurden inzwischen erst vereinzelt für verschiedene Graphenprobleme Algorithmen für MIMD-Rechner entwickelt. Daher wird in dieser Arbeit das Hauptgewicht auf Algorithmen für MIMD-Rechner gelegt.

Zuerst werden in Kapitel 2 die zum Verständnis der Algorithmen notwendigen graphentheoretischen Grundlagen dargestellt und anschließend in Kapitel 3 die verschiedenen parallelverarbeitenden Rechnerstrukturen beschrieben. Insbesondere werden die parallelen Verarbeitungsmöglichkeiten der CRAY

X-MP/22 vorgestellt. Dieses Rechnersystem der Firma CRAY RESEARCH ist ein Doppelprozessorsystem, das einerseits die gleichzeitige Skalar-/Vektorverarbeitung unabhängiger Jobs und andererseits die gleichzeitige Skalar-/Vektorverarbeitung unabhängiger Tasks eines Jobs ermöglicht. Dadurch kann der Parallelismus auf mehreren Ebenen gleichzeitig ausgenutzt werden.

In Kapitel 4 werden einige in der Literatur vorgestellte Graphenalgorithmen für MIMD-Rechner beschrieben, die in einem Pseudocode, der an PASCAL angelehnt ist, dargestellt werden. Dabei handelt es sich um Algorithmen für bewertete Graphen. In einem ersten Teil werden Algorithmen zur Bestimmung kürzester Wege und für die Bestimmung eines minimalen spannenden Baumes in einem Graphen vorgestellt. In einem weiteren Abschnitt werden ein exakter Algorithmus nach dem Branch-and-Bound-Verfahren und zwei Näherungsalgorithmen zur Lösung des Traveling Salesman Problems formuliert.
Ein Teil dieser Algorithmen ist auf der CRAY X-MP/22 in FORTRAN implementiert und auf seine Eignung hinsichtlich der Parallelisierung auf zwei Prozessoren untersucht worden. Die erreichten Ergebnisse verschiedener Testläufe mit unterschiedlichen Eingabedaten sind in Kapitel 5 dokumentiert.

2.1 GRUNDBEGRIFFE

Ein Graph dient als mathematisches Hilfsmittel zur Darstellung von zweistelligen Relationen. Dabei muß berücksichtigt werden, daß die Relationen symmetrisch oder nicht symmetrisch sein können. Daraus ergibt sich eine Unterteilung in ungerichtete und gerichtete Graphen.
Die Bezeichnungen der folgenden Definitionen, Sätze und Beispiele sind aus [DöMü,73] und [Möh,84] entnommen.

2.1.1 Ungerichtete Graphen

Symmetrische Relationen kann man mit Hilfe von ungerichteten Graphen darstellen, die auf verschiedene Arten definiert werden können. In der folgenden Definition werden Kanten als Paare von Knoten erklärt.

Definition 2.1

Ein *ungerichteter Graph* G = (V,E) besteht aus einer endlichen, nichtleeren Menge V von *Knoten* (vertices) und einer Menge E von ungeordneten Paaren e = (u,v), den *Kanten* (edges) von G.
Eine Kante e $\in$ E verbindet ihre zwei *Endpunkte* u und v $\in$ V (u = v möglich). Diese beiden Endpunkte nennt man *adjazent* zueinander. Der Knoten u bzw. v und die Kante e heißen *inzident*. Zwei Kanten sind *adjazent*, wenn sie einen gemeinsamen Endknoten besitzen.

Abb. 1 zeigt einen ungerichteten Graphen G.

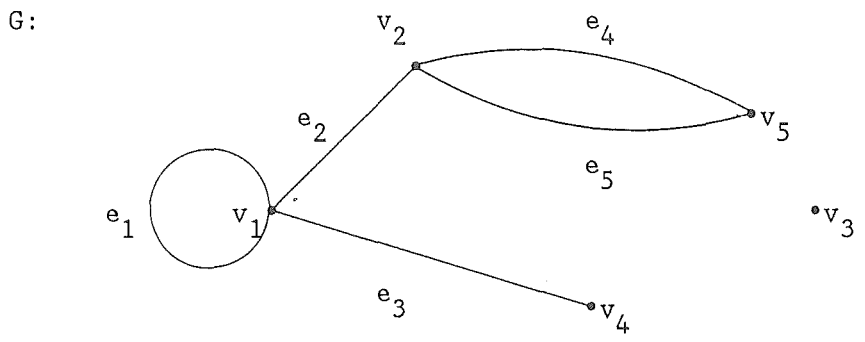


Abb. 1. Ungerichteter Graph

Im folgenden werden weitere Begriffe vorgestellt und anhand von Abb. 1 erläutert.

- *Parallele Kanten* sind Kanten mit gleichen Endpunkten (e_4, e_5).
- Eine *Schleife* (Schlinge, loop) ist eine Kante, die nur mit einem Knoten inzident ist (e_1).
- Ein *isolierter Knoten* ist ein Knoten, der mit keiner Kante inzident ist (v_3).
- Der *Grad* d(v) des Knotens v ist die Anzahl seines Auftretens als Endpunkt einer Kante. Falls keine Schleifen mit v inzidieren, ist d(v) gleich der mit v inzidenten Kanten ($d(v_1) = 4$, $d(v_2) = 3$, etc.).

Definition 2.1 erlaubt das Auftreten von Schleifen und parallelen Kanten in einem ungerichteten Graphen. In der Praxis gibt es Fälle, die diese allgemeine Definition benötigen, aber im folgenden werden, wenn nicht explizit erwähnt, nur Graphen betrachtet, die solche Kanten nicht zulassen.

Definition 2.2

Ein Graph ohne Schleifen und parallele Kanten wird als *einfacher Graph* bezeichnet.

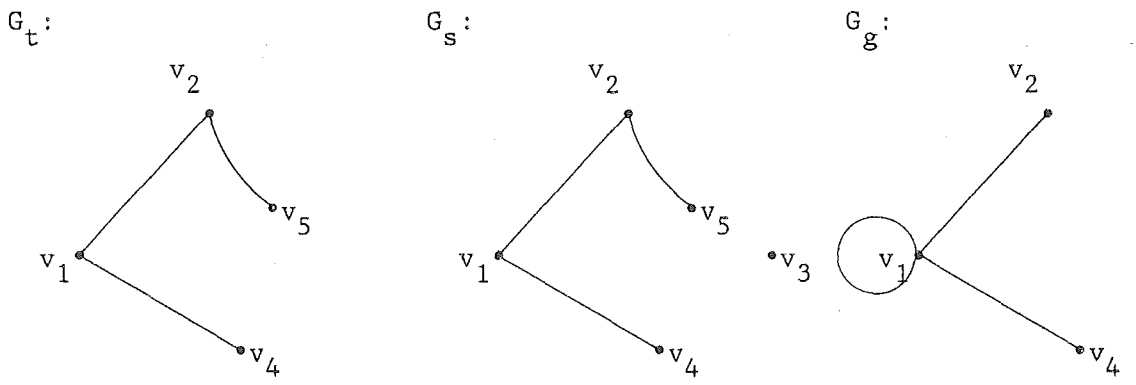
Definition 2.3

Ein Graph $G = (V,E)$ heißt *bewertet*, wenn jeder Kante $e \in E$ eine reelle Zahl $g(e)$ als *Gewicht* zugeordnet ist.

Definition 2.4

Ein Graph $G_1 = (V_1,E_1)$ heißt *Teilgraph* von $G = (V,E)$, wenn $V_1 \subseteq V$ und $E_1 \subseteq E$ ist. Ein Teilgraph G_1 von G heißt *spannender Teilgraph* von G, wenn $V_1 = V$. Enthält E_1 alle Kanten aus E, deren Endpunkte in V_1 liegen, so sagt man, daß G_1 von V_1 in G *aufgespannt* wird, oder daß G_1 ein *gesättigter Teilgraph* von G ist.

In Abb. 2 wird ein Teilgraph, ein spannender Teilgraph und ein gesättigter Teilgraph des Graphen G aus Abb. 1 dargestellt.



$V_t = \{v_1,v_2,v_4,v_5\}$ $E_t = \{e_2,e_3,e_5\}$

$V_s = \{v_1,v_2,v_3,v_4,v_5\}$ $E_s = \{e_2,e_3,e_5\}$

$V_g = \{v_1,v_2,v_4\}$ $E_g = \{e_1,e_2,e_3\}$

Abb. 2. Teilgraph G_t, spannender Teilgraph G_s und gesättigter Teilgraph G_g

Eine der elementaren Eigenschaften, die ein Graph besitzen kann, ist der Zusammenhang. Die Erläuterung der Begriffe zusammenhängender Graph und Zusammenhangskomponenten eines Graphen sind notwendig für die Lösung verschiedener graphentheoretischer Probleme.

Definition 2.5

Eine *Kantenfolge* von v_1 nach v_n im Graphen G ist eine endliche Folge von Kanten (v_1,v_2), (v_2,v_3), ..., (v_{n-1},v_n), wobei je zwei aufeinanderfolgende Kanten einen gemeinsamen Endpunkt haben.
Die Kantenfolge ist *offen*, wenn $v_1 \neq v_n$ ist. Ist $v_1 = v_n$, so ist die Kantenfolge *geschlossen*.
Ein *Kantenzug* ist eine Kantenfolge, in der alle Kanten voneinander verschieden sind.
Ein *Weg* ist eine offene Kantenfolge, in der alle Knoten v_1, ..., v_n verschieden sind.

In Abb. 3 bilden die Kanten e_2, e_1, e_1, e_4 eine offene Kantenfolge, jedoch keinen Kantenzug. Die Kanten e_2, e_4, e_5, e_6 bilden einen Kantenzug aber keinen Weg. Die Kanten e_2, e_1, e_3, e_7, e_8 bilden einen Weg.

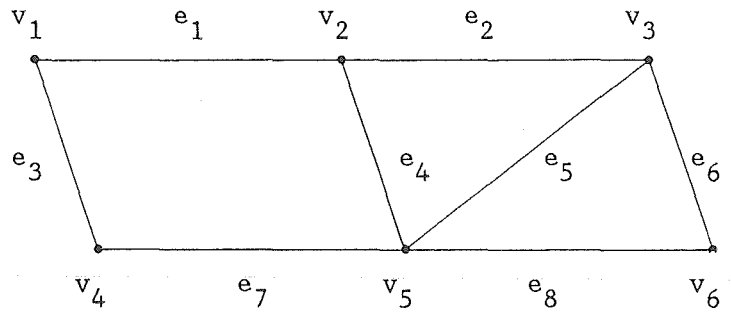


Abb. 3. Ungerichteter Graph

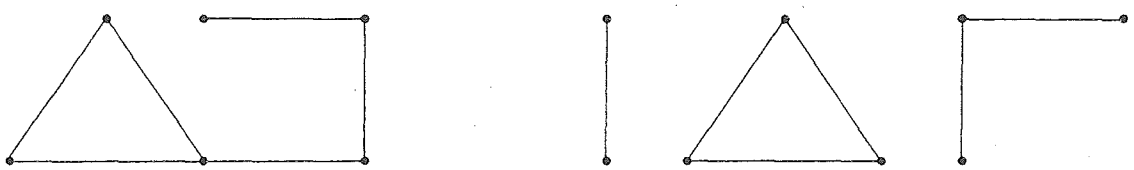
Definition 2.6

Ein *Kreis* ist ein geschlossener Kantenzug, bei dem die Knoten v_1, v_2, ..., v_{n-1} alle verschieden sind.

In Abb. 3 bilden die Kanten e_2, e_1, e_3, e_7, e_5 einen Kreis.

Definition 2.7

Ein Graph G heißt *zusammenhängend*, wenn jeweils zwei beliebige Knoten durch einen Weg in G verbunden sind (siehe Abb. 4).



a) b)

Abb. 4. a) Zusammenhängender Graph, b) Nicht zusammenhängender Graph

Definition 2.8

Sei G = (V,E) ein ungerichteter Graph.
C(v) := {u ∈ V | Es ex. ein Weg von u nach v in G} heißt *Zusammenhangskomponente* von v bez. G für v ∈ V.
Die Mengen C(v), v ∈ V, bilden eine disjunkte Zerlegung von V in die *Zusammenhangskomponenten* von G.

Der Graph in Abb. 4 b) hat 3 Zusammenhangskomponenten.

2.1.2 Gerichtete Graphen

Wenn bei praktischen Anwendungen der Graphentheorie die Relation zwischen den Objekten, die durch einen Graphen dargestellt wird, nicht symmetrisch ist, muß man den Kanten eine Richtung zuweisen.

Definition 2.9

Ein *gerichteter Graph (Digraph)* D = (V,E) besteht aus einer endlichen, nichtleeren Menge V von *Knoten* und einer Menge E von geordneten Paaren e = (u,v) von *Kanten* von D.
Jede Kante e $\in$ E hat einen *Anfangsknoten* u und einen *Endknoten* v aus V (u = v möglich). Der Knoten v heißt *adjazent* zum Knoten u. Der Knoten u bzw. v und die Kante e heißen *inzident*.

Abb. 5 zeigt einen gerichteten Graphen.

Auch hier werden analog zu Kapitel 1.1 einige Begriffe erklärt und anhand von Abb. 5 erläutert.

- *Parallele Kanten* sind Kanten mit gleichen Anfangs- und Endknoten (e_7, e_8).
- *Antiparallele Kanten* sind parallele Kanten, die entgegengesetzt gerichtet sind (e_1, e_2).
- Der *Innengrad* $d_{in}(v)$ eines Knotens ist gleich der Anzahl der eingehenden Kanten ($d_{in}(v_1) = 2$, $d_{in}(v_2) = 2$, $d_{in}(v_3) = 1$, etc.).
- Der *Außengrad* $d_{out}(v)$ eines Knotens ist gleich der Anzahl der ausgehenden Kanten ($d_{out}(v_1) = 1$, $d_{out}(v_2) = 1$, $d_{out}(v_3) = 4$, etc.).
- Der *Grad* d(v) eines Knotens ist die Summe $d_{in}(v) + d_{out}(v)$ ($d(v_1) = 3$, $d(v_2) = 3$, $d(v_3) = 5$, etc.).

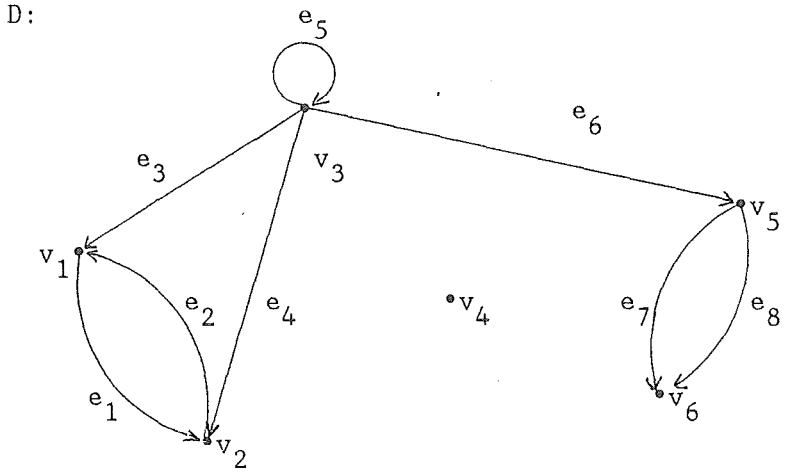


Abb. 5. Gerichteter Graph

In jedem gerichteten Graphen G gilt $\sum_{v \in V} d_{in}(v) = \sum_{v \in V} d_{out}(v) = |E|$, da jede Kante genau einmal in $\sum_{v \in V} d_{in}(v)$ und genau einmal in $\sum_{v \in V} d_{out}(v)$ auftritt.

Isolierte Knoten und *Schleifen* lassen sich analog wie bei ungerichteten Graphen beschreiben.
In Abb. 5 ist v_4 ein isolierter Knoten und e_5 eine Schleife.

Ein *gerichteter einfacher Graph* ist analog zu Definition 2.2, ein *gerichteter bewerteter Graph* analog zu Definition 2.3 definiert. Die Definitionen der Begriffe *gerichteter Teilgraph*, *gerichteter spannender Teilgraph* und *gerichteter gesättigter Teilgraph* erfolgen analog zu Definition 2.4.
Auch bei gerichteten Graphen sind, wenn nicht anders vorausgesetzt, einfache Graphen gemeint.

Definition 2.10

Ein Digraph $D = (V,E)$, mit v_i, $v_j \in V$ heißt *symmetrisch*, wenn mit $(v_i,v_j) \in E$ stets auch (v_j,v_i) in E liegt.

Entsprechende Begriffsbildungen wie bei ungerichteten Graphen sind *gerichtete Kantenfolge*, *gerichteter Kantenzug*, *gerichteter Weg* und *gerichteter Kreis*.
Ein Digraph $D = (V,E)$ heißt *azyklisch*, falls er keine gerichteten Kreise (*Zyklen*) enthält.

Für die Problemformulierung der kostenminimalen Verbindung zwischen zwei Knoten in einem gerichteten Graphen, ist folgende Definition über die Länge eines gerichteten Weges notwendig.

Definition 2.11

Sei $D = (V,E)$ ein Digraph, dessen Kanten $e = (v_i,v_j)$ mit den Gewichten $g(e) = g((v_i,v_j))$ bewertet sind.
Ist $W(v_1,v_n) = \{(v_1,v_2), \ldots, (v_{n-1},v_n)\}$ ein Weg von $v_1 \in V$ nach $v_n \in V$, dann heißt $\sum_{e \in W} g(e)$ die *Länge des Weges* W.

Ist $M(v_1,v_n) = \{W_i \mid W_i = W_i(v_1,v_n)\}$, so heißt $W_i \in M(v_1,v_n)$ *kürzester Weg* von v_1 nach v_n, wenn für alle Wege $W_j \in M(v_1,v_n)$ mit $j \neq i$ gilt:

$$\sum_{e \in W_i} g(e) \leq \sum_{e \in W_j} g(e).$$

Bemerkung: Definition 2.11 gilt auch für ungerichtete Graphen, wenn diese als symmetrische gerichtete Graphen angesehen werden.

Wesentliche Unterschiede zwischen gerichteten und ungerichteten Graphen ergeben sich bei der Definition der Zusammenhangseigenschaft.
Durch Weglassen der Richtungen entsteht aus einem Digraphen der *zugeordnete ungerichtete Graph*.

Definition 2.12

Ein Digraph heißt *(schwach) zusammenhängend*, falls der zugeordnete ungerichtete Graph zusammenhängend ist.

Definition 2.13

Ein Digraph heißt *stark zusammenhängend*, falls es zu je zwei Knoten u, v einen gerichteten Weg von u nach v und einen gerichteten Weg von v nach u gibt.

Definition 2.14

Sei D = (V,E) ein Digraph.
$C(v) := \{u \in V \mid$ Es ex. in D gerichtete Wege von v nach u und von u nach v$\}$ heißt *starke Zusammenhangskomponente* von v bez. D, für $v \in V$.
Die Mengen C(v) ($v \in V$) bilden eine disjunkte Zerlegung von V in die *starken Zusammenhangskomponenten* von D.

2.1.3 Bäume

Sowohl bei theoretischen Untersuchungen als auch bei praktischen Anwendungen spielt eine Klasse besonders einfach strukturierter Graphen eine große Rolle. Dabei handelt es sich um die Graphen, in denen es keine Kreise gibt.

Definition 2.15

Sei G = (V,E) ein ungerichteter Graph.
G heißt *kreisfrei* :<=> G enthält keine Kreise.
G heißt *Baum* :<=> G ist zusammenhängend und kreisfrei.

Definition 2.16

Sei D = (V,E) ein Digraph. Ein Knoten v in D heißt *Wurzel*, falls jeder Knoten von v aus auf einem gerichteten Weg erreichbar ist.
D heißt *gerichteter Baum*, falls er eine Wurzel hat und der zugeordnete ungerichtete Graph ein Baum ist.

Folglich ist in einem gerichteten Baum ein Knoten mit Innengrad 0 als Wurzel ausgezeichnet, und alle anderen Knoten haben Innengrad 1. Solche Bäume werden auch *Wurzelbäume* genannt.
Die Knoten des gerichteten Baumes mit Außengrad 0 heißen *Endknoten (Blätter)*, alle anderen Knoten werden als *innere Knoten* bezeichnet. Eine Ansammlung von Bäumen wird *Wald* genannt.

Definition 2.17

Gegeben sei ein gerichteter Baum mit u,v und v' aus V.
Wenn (u,v) in E enthalten ist, dann heißt u *Vater* von v und v *Sohn* von u. Gibt es einen Weg von u nach v', so ist u ein *Vorgänger* von v' und v' ein *Nachfolger* von u. Ein Knoten u mit allen seinen Nachfolgern wird *Teilbaum* genannt. Der Knoten u heißt *Wurzel des Teilbaums*.
Das *Niveau* (level) *eines Knotens* v in einem Baum ist die Länge des Weges von der Wurzel nach v. Die *Höhe eines Knotens* v in einem Baum ist die Länge des längsten Weges von v zu einem Blatt. Die *Höhe eines Baumes* ist die Höhe der Wurzel.

Abb. 6 zeigt einen gerichteten Baum.

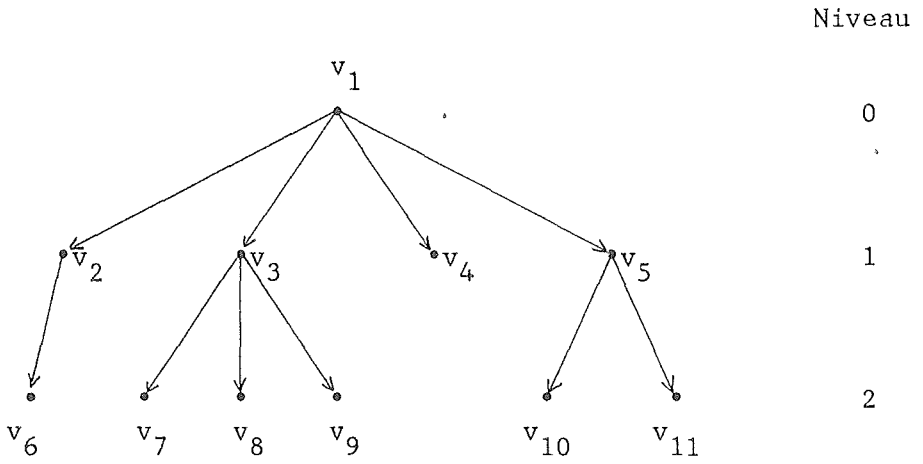


Abb. 6. Gerichteter Baum

Beispiel 2.1

Die Wurzel des Baumes in Abb. 6 ist der Knoten v_1. Er liegt auf dem Niveau 0. Die Höhe des Baumes ist 2. Der Knoten v_4 auf dem Niveau 1 und die Knoten $v_6, v_7, \ldots, v_{11}$ auf dem Niveau 2 sind die Blätter des Baumes. v_5 ist Vater von v_{10} und v_9 ist Sohn von v_3. Die Knoten v_3, v_7, v_8 und v_9 bilden einen Teilbaum mit Wurzel v_3.

Eine spezielle Klasse der gerichteten Bäume ist die Klasse der Binärbäume [AHU,74], die für viele Graphenalgorithmen nützlich sind.

Definition 2.18

Ein *Binärbaum* ist ein gerichteter Baum, in dem jeder Knoten höchstens zwei Söhne hat. Jeder Sohn eines Knotens wird als linker oder rechter Sohn bezeichnet.
Ein Binärbaum heißt *vollständig*, wenn für eine ganze Zahl k gilt:
Jeder Knoten auf einem Niveau kleiner als k hat zwei Söhne und jeder Knoten auf dem Niveau k ist ein Blatt. Ein vollständiger Binärbaum der Höhe k hat genau $2^{k+1}-1$ Knoten.

Definition 2.19

Sei G = (V,E) ein ungerichteter Graph, D = (V,E) ein Digraph und D_u der zugeordnete ungerichtete Graph.

- Ist der Teilgraph T von G zugleich spannender Teilgraph und ein Baum, so heißt T *spannender Baum* von G.
- Ist der Teilgraph T_s von D_u zugleich spannender Teilgraph und ein Baum, so heißt T_s *spannender Baum* von D.
- Ist der gerichtete Teilgraph T_g von D zugleich ein gerichteter spannender Teilgraph und ein gerichteter Baum, so heißt T_g *gerichteter spannender Baum* von D.

Abb. 7 zeigt einen spannenden Baum und einen gerichteten spannenden Baum des Graphen D.

Satz: 2.1

Ein Graph G = (V,E) besitzt genau dann einen spannenden Baum, wenn G zusammenhängend ist.

Der Beweis des Satzes kann in [DöMü,73] nachgelesen werden.

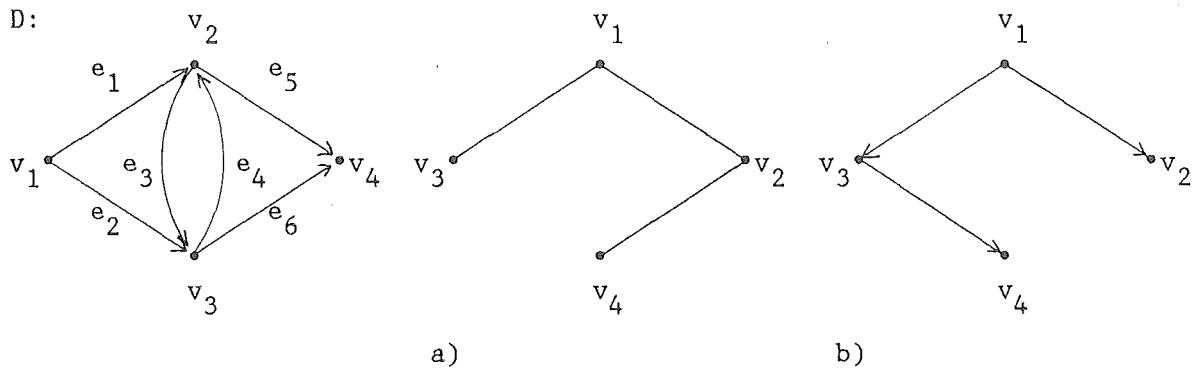


Abb. 7. a) Spannender Baum, b) Gerichteter spannender Baum

Bemerkung: Ein spannender Baum ist nur für zusammenhängende Graphen definiert, da ein Baum immer zusammenhängend ist und ein nicht zusammenhängender Graph keinen zusammenhängenden spannenden Teilgraph hat. Jede Zusammenhangskomponente eines nicht zusammenhängenden Graphen hat einen spannenden Baum. Folglich hat ein nicht zusammenhängender Graph mit k Zusammenhangskomponenten einen *spannenden Wald*, der aus k spannenden Bäumen besteht.

In vielen Problemen der Graphentheorie werden spannende Bäume eines Graphen untersucht, insbesondere werden minimale spannende Bäume von bewerteten Graphen bestimmt.

Definition 2.20

Sei G = (V,E) ein ungerichteter endlicher zusammenhängender Graph. Jede Kante e $\in$ E sei mit g(e) bewertet.
T = (V,E') heißt *minimaler spannender Baum*, wenn
(i) T ein spannender Baum ist und
(ii) $\sum_{e \in E'} g(e) \leq \sum_{e \in E''} g(e)$
für jeden anderen spannenden Baum T' = (V,E''), mit E', E'' $\subseteq$ E.

2.2 DARSTELLUNG VON GRAPHEN

Es gibt zwei Möglichkeiten die Relation zwischen den Knoten eines Graphen in einer für die Speicherung in Rechenanlagen geeigneten Art und Weise darzustellen: Matrizen und Listen.
Die Bezeichnungen der folgenden Definitionen lehnen sich an [DöMü,73] und [Möh,84] an. Das Beispiel ist aus [AHU,74] entnommen.

2.2.1 Matrizen

Als Matrix kann man entweder die Inzidenz von Knoten und Kanten oder die Adjazenz von Knoten darstellen.

Definition 2.21

Sei G = (V,E) ein ungerichteter Graph. Die Knoten und Kanten seien mit einer festen Numerierung versehen, also $V = \{v_1, \ldots, v_n\}$ und $E = \{e_1, \ldots, e_m\}$.
Die *Inzidenzmatrix* B zur gegebenen Numerierung von V und E ist eine n × m Matrix, die definiert ist durch

$$b_{ij} = \begin{cases} 1, \text{ wenn } v_i \text{ und } e_j \text{ inzident sind} \\ 0, \text{ sonst} \end{cases}$$

Die Spalten von B entsprechen den Kanten des Graphen. Jede Spalte enthält genau zwei Einsen, sonst Nullen. Die Zeilen der Matrix entsprechen den Knoten des Graphen. Die Anzahl der Einsen pro Zeile gibt den Grad des Knotens an.
Der Speicherplatzbedarf einer Inzidenzmatrix beträgt O(|V|*|E|).

Beispiel 2.2

Die Inzidenzmatrix des ungerichteten Graphen aus Abb. 3 sieht wie folgt aus:

$$B = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Definition 2.22

Sei D = (V,E) ein gerichteter Graph mit numerierter Knoten- und Kantenmenge, so ist die *Inzidenzmatrix* B für v_i, $w \in V$ definiert durch

$$b_{ij} = \begin{cases} 1, \text{ wenn } e_j = (v_i, w) \in E \\ -1, \text{ wenn } e_j = (w, v_i) \in E \\ 0, \text{ sonst} \end{cases}$$

Beispiel 2.3

Die folgende Inzidenzmatrix ist eine Darstellung des gerichteten Graphen aus Abb. 7.

$$B = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 & 1 & 0 \\ 0 & -1 & -1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & -1 \end{bmatrix}$$

Definition 2.23

Sei $X = (V,E)$ ein ungerichteter oder gerichteter Graph mit $V = \{v_1, \ldots, v_n\}$.
Die *Adjazenzmatrix* A eines Graphen ist eine n × n Matrix mit

$$a_{ij} = \begin{cases} 1, \text{ wenn } v_j \text{ adjazent zu } v_i \text{ ist} \\ 0, \text{ sonst} \end{cases}$$

Ungerichtete Graphen haben eine symmetrische Adjazenzmatrix. Es genügt, nur die Elemente unterhalb der Hauptdiagonalen zu betrachten. Damit läßt sich bei einem Graphen mit vielen Knoten ein erheblicher Teil des Speicherplatzes einsparen.

Beispiel 2.4

Der gerichtete Graph D aus Abb. 7 läßt sich mit Hilfe der Adjazenzmatrix wie folgt darstellen.

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Zusätzlich zu der beschriebenen Adjazenzmatrix können für bewertete ungerichtete oder gerichtete Graphen die Gewichte mit in eine *bewertete Adjazenzmatrix* gebracht werden. Anstelle der Einsen in einer nicht bewerteten Adjazenzmatrix werden hier die Kantenbewertungen in die Matrix eingetragen.

Der Speicherplatzbedarf einer Adjazenzmatrix ist $O(|V|^2)$. Ein Vorteil der Speicherung von Graphen als Adjazenzmatrizen ist der direkte Zugriff auf die Elemente (i,j) in der Matrix. Dies ist vorteilhaft für Algorithmen, die Kanten einfügen, löschen oder oft feststellen müssen, ob bestimmte Kanten im Graph existieren oder nicht.

2.2.2 Listen

Der Platzbedarf einer Adjazenzmatrix A eines Graphen, der relativ zu seiner Knotenzahl n wenige Kanten besitzt, ist verhältnismäßig groß. Die Matrix A ist dann bez. der Speicherauslastung *dünnbesetzt* (schwachbesetzt). In solchen Fällen sollte wegen der möglichen Speicherplatzeinsparung die Darstellung des Graphen durch Listen erfolgen. Die Kanten des Graphen werden als Paare von Knoten aufgelistet.

Eine Möglichkeit der Speicherung ist die Implementierung durch zwei Felder $g = (g_1, \ldots, g_m)$ und $h = (h_1, \ldots, h_m)$. Jedes Feldelement ist eine Kno-

tenbezeichnung und die i-te Kante eines Graphen geht von Knoten g_i nach Knoten h_i ([RND,77], S.321).
Der Digraph D aus Abb. 7 wird beispielsweise durch die beiden folgenden Felder repräsentiert:

g = (1, 1, 2, 3, 2, 3)
h = (2, 3, 3, 2, 4, 4)

Als eine andere Darstellungsart kann man Adjazenzlisten wählen.

Definition 2.24

Die *Adjazenzliste* für einen Knoten $v \in V$ ist eine Liste aller Knoten $w \in V$, die adjazent zu v sind.

Die Adjazenzlisten der Knoten des Graphen D aus Abb. 7 sehen z.B. wie folgt aus.

v	Adj(v)
1:	2, 3
2:	3, 4
3:	2, 4
4:	leere Liste

Der Speicherplatzbedarf bei Darstellung eines Graphen durch Adjazenzlisten beträgt $O(|V|+|E|)$. Diese Art der Speicherung ist vorteilhaft für Operationen, die alle Kanten durchsuchen, die von einem Knoten ausgehen.

3.0 PARALLELVERARBEITENDE RECHNERSTRUKTUREN

3.1 KLASSIFIKATION

Bei der Behandlung von Parallelrechnersystemen kann man nach Parallelismus innerhalb des Befehlsstroms und nach Parallelismus innerhalb des Datenstroms klassifizieren. Mit *Befehlsstrom* (instruction stream) ist eine Befehlssequenz gemeint, die in einem Prozessor ausgeführt wird. Ein *Datenstrom* (data stream) ist eine Folge von Operanden, die in einem Prozessor verarbeitet werden. Einer der ersten, der sich mit dem Problem der Rechnerklassifizierung eingehend beschäftigte, war Michael J. Flynn [Fly,66]. Seine Klassifizierung ist am weitesten verbreitet, kann jedoch Überschneidungen zwischen den Klassen nicht ausschließen. Nach der obigen Unterscheidung zwischen Befehlsstrom und Datenstrom ergeben sich vier Klassen.

Die Klasse der Single Instruction Stream Single Data Stream (SISD) Rechner umfaßt alle konventionellen Rechner des von-Neumann-Typs.

In die Klasse der Single Instruction Stream Multiple Data Stream (SIMD) Rechner gehören die Arrayprozessoren sowie die Vektor- und Pipeline-Rechner.

Die Klasse der Multiple Instruction Stream Single Data Stream (MISD) Rechner muß als faktisch leer angesehen werden, da zur Berechnung eines einzigen Datenwertes bereits ein ganzer Befehlsstrom notwendig wäre.

Durch Kombination von parallelen Befehlsströmen und parallelen Datenströmen erhält man die Klasse der Multiple Instruction Stream Multiple Data Stream (MIMD) Rechner. Zu dieser Kategorie gehören die Multiprozessorsysteme, aber auch komplexere Array-Strukturen mit mehreren Kontrolleinheiten.

Abb. 8 gibt einen Überblick über die verschiedenen Rechnerstrukturen.

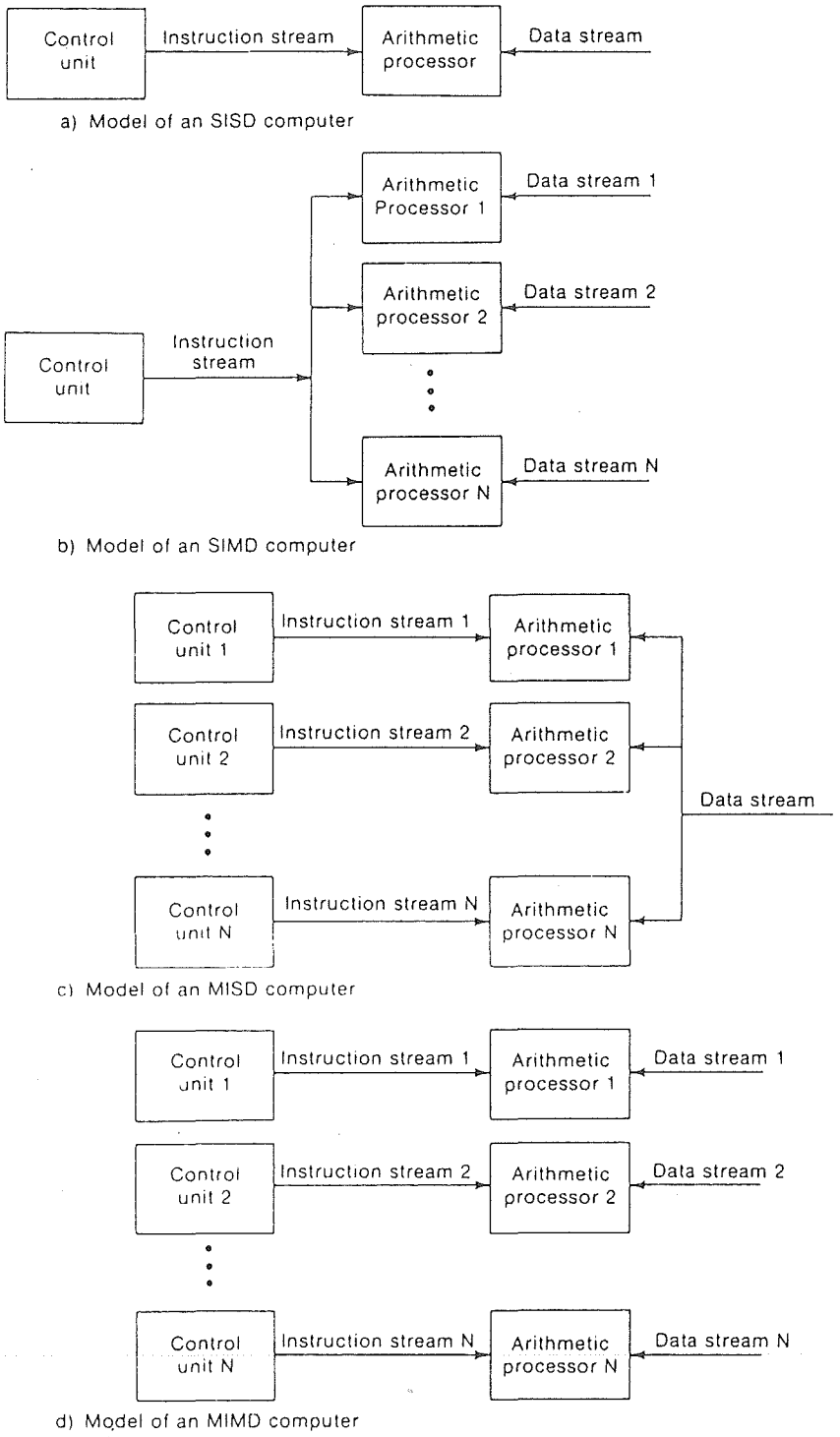


Abb. 8. Rechnermodelle

Nach dieser Klassifizierung verbleiben für die verschiedenen Rechnerstrukturen nur zwei Klassen, die über die von-Neumann-Architektur hinausgehen. Die SIMD- und MIMD-Rechner unterscheiden sich wesentlich dadurch, wie sie Parallelität in ihren Operationen erzielen. Die SIMD-Rechner sind gut geeignet für Berechnungen, die in eine Folge von Vektoroperationen aufgeteilt werden können. Die MIMD-Rechner haben die Fähigkeit Prozesse auf mehreren Prozessoren parallel zu bearbeiten [Sto,75], [Hoß,84].

3.1.1 SIMD-Rechner

Die Klasse der SIMD-Rechner ist aus heutiger Sicht die wichtigste Kategorie der sogenannten *Parallelprozessoren*. Dies gilt sowohl im Hinblick auf die auf dem Markt befindlichen "Supercomputer" mit der Architektur der Vektor- bzw. Pipelinerechner wie CRAY oder CYBER 205, als auch in Hinblick auf die algorithmischen Entwicklungsarbeiten. In dieser Klasse sind alle Rechner mit sogenannten *Vektorinstruktionen* enthalten.
In diese Kategorie fallen aber auch die Array-Rechner mit einer Kontrolleinheit und einem entsprechenden Array von Prozessorelementen, die zur gleichen Zeit alle die gleiche Instruktion ausführen müssen, wie beispielsweise der ILLIAC IV [Hoß,81].
Zu den Array-Rechnern gehören auch die Systeme mit einem systolischen Array. Das Grundprinzip eines systolischen Arrays ist in Abb. 9 dargestellt.

Ersetzt man das einzelne Prozessorelement des konventionellen Rechners, der in der Abbildung dargestellt wird, durch ein Array von Prozessorelementen, so kann ohne Erhöhung der Bandbreite des Speichers ein höherer Durchsatz des Systems erreicht werden. Die Funktion des Speichers entspricht hier dem menschlichen Herzen: Er pumpt die Daten durch das Array von Prozessorelementen. Voraussetzung für die Leistungsfähigkeit systolischer Arrays ist, daß jedes Element des Datenstromes, das den Speicher verläßt, von jedem Prozessorelement effektiv genutzt werden kann. Diese Voraussetzung trifft auf viele rechenintensive Probleme zu, da viele Ope-

rationen wiederholt bei jedem Datenelement durchgeführt werden müssen [HwBr,84].

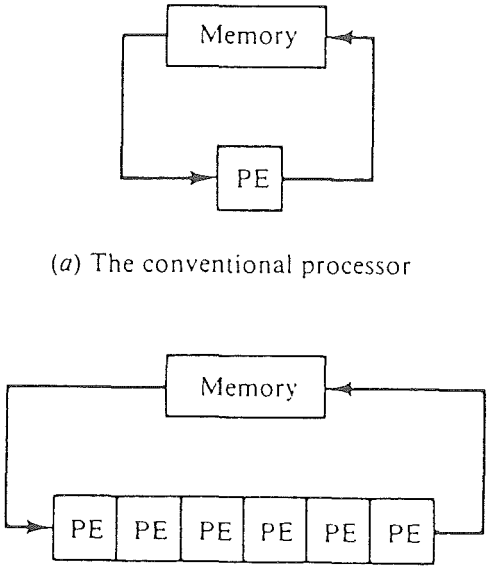


Abb. 9. Das Konzept eines systolischen Prozessor-Arrays

Damit ein Problem für die Berechnung auf einem SIMD-Rechner geeignet ist, müssen folgende Eigenschaften erfüllt sein:

1. Die Berechnung muß durch Vektorinstruktionen ausgedrückt werden können, so daß durch viele gleiche parallele Operationen mit verschiedenen Daten Rechenzeit gespart werden kann.
2. Zwischen den Prozessoren muß ein schneller Datenaustausch möglich sein.
3. Operanden, die gleichzeitig bearbeitet werden sollen, müssen gleichzeitig aus dem Speicher geholt werden können.

Wenn eine der drei Bedingungen für ein Programm nicht erfüllt ist, dann läuft die Berechnung auf einem SIMD-Rechner im wesentlichen sequentiell

ab. Die Ausnutzung der SIMD-Fähigkeit setzt voraus, daß nur eine bestimmte Klasse von Berechnungen durchgeführt wird, da nicht alle Berechnungen die oben genannten Punkte erfüllen.
Im Gegensatz zu den MIMD-Rechnern sind die Prozessoren durch den gemeinsamen Takt während einer Programmausführung Befehl für Befehl automatisch synchronisiert [Sto,75].

3.1.2 MIMD-Rechner

Multiprozessorsysteme bestehen aus mehreren CPUs, die sowohl über einen privaten als auch über einen zentralen Speicher verfügen können. Man unterscheidet Systeme, in denen die Prozessoren stark gekoppelt (*tightly coupled*) oder schwach gekoppelt (*loosely coupled*) sind [Gil,81].
In einem stark gekoppelten System haben die Prozessoren über einen Schaltungsmechanismus Zugriff auf einen gemeinsamen Speicher (*memory sharing*). Wegen der Kosten für den Schaltungsmechanismus, die bei großen Systemen sehr schnell ansteigen, ist die Anzahl der Prozessoren bei diesen Systemen beschränkt. In diese Kategorie gehören der C.mmp der Carnegie-Mellon Universität mit 16 Prozessoren und der HEP von Denelcor mit bis zu 8 Pipeline-Prozessoren.
Im Gegensatz dazu hat jeder Prozessor in einem schwach gekoppelten System einen eigenen lokalen Speicher. Die Kommunikation ist hier auf den Austausch von Botschaften über eine gemeinsame Kommunikationseinrichtung (*message switching*) beschränkt. Ein Beispiel für diese Systeme ist der Cm* der Carnegie-Mellon Universität mit 50 Prozessoren [QuDe,84].
Der Parallelismus bei MIMD-Rechnern besteht in der parallelen Bearbeitung mehrerer Prozesse auf unabhängigen Prozessoren. Ein *Prozeß* ist als Programm(teil) in Ausführung definiert.
Bei der Anpassung einer Berechnung an einen MIMD-Rechner ist die Aufteilung in verschiedene Prozesse, die parallel ablaufen können, problematisch. Die Unterteilung wird schwieriger, wenn mehr Prozessoren zur Verfügung stehen und diese effizient genutzt werden sollen. Weitere Probleme bestehen in der Synchronisierung der Berechnungen auf den verschiedenen Prozessoren und in der Zuweisung der Prozessoren an die Prozesse [Sto,75].

3.2 BESCHREIBUNG DER CRAY X-MP/22

Die CRAY X-MP/22 ist eine Erweiterung der CRAY-1 zu einem Multiprozessorsystem. Sie besitzt zwei Zentraleinheiten, die auf einem gemeinsamen Speicher arbeiten (stark gekoppelt).
Die Parallelität von Programmen wird auf der CRAY X-MP in zweierlei Hinsicht ausgenutzt: durch *Vektorisierung* und durch *Multitasking*.
Da beide Prozessoren selbständig arbeiten können, erlaubt die Fähigkeit der Vektorverarbeitung eine Einordnung in die Klasse der SIMD-Rechner. Betrachtet man dagegen die Möglichkeit des Multitasking, bei der beide Prozessoren für ein Programm ausgenutzt werden können, so ergibt sich eine Zuordnung zur Klasse der MIMD-Rechner.

3.2.1 Vektorverarbeitung

In vielen Anwendungen des technisch-wissenschaftlichen Bereiches müssen große Datenmengen als Felder oder Vektoren von Gleitpunktzahlen verarbeitet werden. Diese Anforderungen waren Ausgangspunkt des Konzepts der Vektorarchitektur.
Die CRAY X-MP bietet wegen der speziellen Vektorarchitektur eine wesentlich höhere Leistungsfähigkeit als konventionelle Rechner.
Viele mathematische Probleme lassen sich von Natur aus durch die Bearbeitung von Vektoren oder Matrizen lösen. Bei der sequentiellen Verarbeitung muß jede Feldkomponente und jedes Matrixelement einzeln angesprochen werden. Dies geschieht programmtechnisch mit Hilfe von Schleifen. Die Vektorverarbeitung hingegen ermöglicht eine Zusammenfassung von Operanden einer arithmetischen oder logischen Operation zu einer Einheit (*Vektor*), die mit einer einzelnen Operation (*Vektoroperation*) verarbeitet werden können [KFA52,85].

3.2.2 Multitasking

Werden mehrere Prozesse durch zwei oder mehr Prozessoren (CPUs) parallel verarbeitet, so spricht man von **Multiprocessing**. Dabei arbeiten alle Prozessoren gleichzeitig, ohne sich gegenseitig zu beeinträchtigen. **Multitasking** ist ein Spezialfall des Multiprocessing, wobei die Prozesse hier spezielle *Tasks* sind, die als Programmschritte oder Unterprogramme definiert werden. Auf der CRAY X-MP ist Multitasking zur Zeit nur in der Programmiersprache FORTRAN möglich. Als selbständige Tasks können momentan nur Unterprogramme erzeugt werden.
Ziel des Multitasking ist es, die Ausführungszeit gegenüber einem Programm, das ohne Multitasking ausgeführt wird, zu reduzieren. Dies heißt aber nicht etwa, daß durch Multitasking die Anzahl der CPU-Zyklen, die für die Ausführung eines Programms notwendig sind, verringert werden. Im Gegenteil, Multitasking bewirkt einen Overhead, der zu einer Erhöhung der CPU-Zeit führt. Dieser Overhead liegt für jeden Aufruf einer Multitasking-Bibliotheksroutine zwischen 1µs und 1ms. Der bei der Ausführung eines Programms insgesamt entstehende Overhead ist abhängig vom Problem und vom Programmierstil.
Bei der Verwendung des Multitasking müssen die Tasks und die Datenstruktur eines Jobs so beschaffen sein, daß die Tasks parallel arbeiten können. Die Zuordnung der Prozessoren auf die Tasks erfolgt nicht automatisch, sondern wird vom Programmierer vorgenommen. Die Verfügbarkeit der Prozessoren wird anhand von Scheduling-Strategien durch das Betriebssystem gesteuert. Damit die Tasks in einer bestimmten Reihenfolge abgearbeitet und gewünschte Resultate auch korrekt erzielt werden können, benötigt man Kommunikations- und Synchronisationsmechanismen zwischen parallelen Tasks [CRAY,222].
Es gibt drei Arten von Multitasking-Bibliotheksroutinen, die dem Programmierer für die parallele Verarbeitung von Tasks zur Verfügung stehen [CRAY,222], [Lars,84]. Sie sind derart implementiert worden, daß sie von FORTRAN-Ebene aus durch CALL-Anweisungen aufgerufen werden können.

- TASK-Routinen
  In einem Programm, das als übergeordnete Task angesehen wird, kann mit dem Aufruf der **TSKSTART**-Routine eine weitere Task gebildet werden, die mit dem Namen des Unterprogramms versehen wird, das als selb-

ständige Task abgearbeitet werden soll. Mit der TSKWAIT-Routine kann die aufrufende Task solange verzögert werden, bis die aufgerufene Task vollständig abgearbeitet ist.

- EVENT-Routinen

  Die EVENT-Routinen ermöglichen die Kommunikation und Synchronisation zwischen den Tasks. Während der Ausführung paralleler Tasks kann es vorkommen, daß die Ergebnisse der Berechnungen einer Task von anderen Tasks benötigt werden. Die das Ergebnis liefernde Task signalisiert das Ende ihrer Berechnung mittels der EVPOST-Routine. Alle Tasks, die auf dieses Ergebnis warten, werden durch den Aufruf der EVWAIT-Routine solange verzögert, bis sie das Signal zur weiteren Ausführung durch die EVPOST-Routine erhalten. Dieses Signal kann durch die EVCLEAR-Routine wieder gelöscht werden.

- LOCK-Routinen

  Die LOCK-Routinen LOCKON und LOCKOFF kontrollieren den sogenannten kritischen Bereich einer Task. Ein *kritischer Bereich* ist ein Programmsegment einer Task, das auf ein gemeinsames Betriebsmittel der Tasks zugreift. Solche kritischen Bereiche sind z.B. Programmsegmente mit Zugriff auf gemeinsamen Speicher, I/O Files oder Unterprogramme. LOCKON und LOCKOFF sorgen dafür, daß zwei oder mehrere Tasks nicht gleichzeitig in ihren jeweiligen kritischen Bereichen tätig sein können, d.h. sie gewährleisten den wechselseitigen Ausschluß von kritischen Bereichen.

4.0 PARALLELE ALGORITHMEN FÜR GRAPHENPROBLEME

Graphentheorie wird für viele Probleme der Technik und Wissenschaft angewendet, wobei praktische Graphenprobleme oft viel Rechenzeit benötigen. Aufgrund der Fortschritte in der VLSI-Technologie wird immer mehr nach effizienten parallelen Graphenalgorithmen geforscht. In manchen Fällen können existierende sequentielle Algorithmen in effiziente parallele Algorithmen umgeformt werden; in anderen Fällen müssen parallele Algorithmen neu entwickelt werden.

Für die hier behandelten parallelen Algorithmen wird angenommen, daß P Prozesse parallel ausgeführt werden können. Die Kommunikation erfolgt über einen gemeinsamen Speicher. Das Erzeugen der Prozesse wird durch die Operation **create** bewerkstelligt. Für die Synchronisierung der Prozesse stehen die beiden Operationen **lock** und **unlock** zur Verfügung. Aus praktischen Gründen wird angenommen, daß die obigen Operationen eine gewisse Zeit zur Ausführung brauchen, wobei **create** mehr Zeit in Anspruch nimmt als **lock** und **unlock**.
Wegen der notwendigen Synchronisation zwischen den Prozessen sind die folgenden parallelen Algorithmen nicht so klar und übersichtlich wie die sequentiellen Algorithmen. Synchronisation ist notwendig bei Zugriff auf globale Variablen, die im gemeinsamen Speicher stehen. Aus Gründen der Übersichtlichkeit werden die folgenden Abkürzungen benutzt:
increment x = (**lock** x; x := x + 1; **unlock** x)
decrement x = (**lock** x; x := x - 1; **unlock** x).

Für alle folgenden graphentheoretischen Algorithmen sei N = |V| und M = |E|.

4.1 KÜRZESTE WEGE

Von großer praktischer Bedeutung für Transport- und Kommunikationsnetzwerke sind Kürzeste-Wege-Probleme.

Für zwei spezielle Arten von kürzeste-Wege-Problemen sind parallele Algorithmen entwickelt worden:

1. Bestimmung des kürzesten Weges von einem festen Knoten (Quelle, source) zu allen anderen Knoten in einem Graphen (*Single-source shortest-path problem*).
2. Bestimmung des kürzesten Weges zwischen allen Knotenpaaren in einem Graphen (*All-pairs shortest-path problem*).

In einem gerichteten Graphen dürfen die Gewichte der Kanten positiv, negativ oder Null sein, solange keine negativ gewichteten Zykel entstehen, da sonst der kürzeste Weg zu einigen Knoten undefiniert ist.
In einem ungerichteten Graphen müssen die Kanten nichtnegative Gewichte haben, da sie in beiden Richtungen durchlaufen werden können.
Sind diese beiden Bedingungen nicht erfüllt, so ist zwar der kürzeste Weg ohne Wiederholung von Knoten definiert, in den hier vorgestellten Verfahren werden jedoch Knotenwiederholungen nicht ausgeschlossen.

4.1.1 Kürzester Weg von der Quelle zu allen anderen Knoten

Für die Bestimmung des kürzesten Weges von der Quelle zu allen anderen Knoten in einem Graphen eignen sich folgende sequentielle Algorithmen zur Parallelisierung auf einem MIMD-Rechner.

1. Der Algorithmus von Moore [Moore,57]
2. Der Algorithmus von Dijkstra [Dijk,59]

Algorithmus von Moore-Pape

Der Algorithmus, den Moore entwickelt hat, ist von Pape [Pape,80] modifiziert worden. Moore benutzt in seinem Algorithmus eine Queue, in die Knoten am Ende eingefügt werden. Der Algorithmus von Pape verwendet ein Deque, d.h. eine Queue, in die Knoten am Anfang und am Ende eingefügt werden können.

Die bewertete Adjazenzmatrix W enthält die Längen der Kanten in dem Graphen. Die Diagonale besteht aus Nullen, und Knotenpaare, zwischen denen keine Kante existiert, haben die Bewertung ∞. DIST[v] gibt die Länge des bisher bekannten kürzesten Weges von der Quelle nach v an, PRED[v] ist der Vorgängerknoten von v auf diesem Weg. Nach Beendigung des Algorithmus enthält das Feld DIST die kürzeste Entfernung von der Quelle zu jedem Knoten im Graphen. Das Feld PRED gibt die Vorgängerknoten jedes Knotens auf dem kürzesten Weg an.
Eine Queue Q enthält die Knoten, von denen aus weiter gesucht werden muß. Sie wird initialisiert mit der Quelle. Solange Q nicht leer ist, wird der Knoten u vom Kopf von Q entnommen und alle Kanten (u,v) untersucht. Wenn DIST[u] + W[u,v] < DIST[v], dann ist ein neuer kürzerer Weg von der Quelle nach v gefunden. DIST[v] wird geändert und der Knoten v wird an das Ende von Q angehängt, falls er nicht schon in Q ist und auch vorher nicht in Q war. Ist er vorher schon in Q gewesen, so wird der Knoten am Kopf von Q eingefügt.
Durch die Implementierung mit einem Deque werden Fehler in den minimalen Entfernungen berichtigt, sobald sie entdeckt worden sind, und nicht weiter fortgesetzt. In der Praxis erhält man mit dieser Version eine geringere CPU-Zeit. Im schlimmsten Fall steigt die benötigte Zeit jedoch exponentiell an [Yoo,83].
Für den Algorithmus von Moore beträgt die Zeitkomplexität im schlimmsten Fall:

$$\sum_{i=1}^{N} (N - 1) * d_i \leq O(M * N).$$

Jeder Knoten wird höchstens N-1 mal in die Queue eingefügt. Wenn ein Knoten aus der Queue entfernt wird, werden alle von diesem Knoten ausgehenden Kanten untersucht. Dabei stellt d_i die Anzahl der von diesem Knoten i ausgehenden Kanten dar. Die auf die for-Anweisung folgende if-Abfrage wird dann höchstens d_i mal ausgeführt.

Der Algorithmus sieht wie folgt aus:

```
for alle u ≠ SOURCE do
   DIST[u] := ∞;
DIST[SOURCE] := 0;
initialisiere Q mit SOURCE;
while Q ist nicht leer do
   begin
      entferne Knoten u am Kopf von Q;
      for jede Kante (u,v) do
         if DIST[v] > DIST[u] + W[u,v] then
            begin
               PRED[v] := u;
               DIST[v] := DIST[u] + W[u,v];
               if v war niemals in Q then
                  füge v am Ende von Q ein;
               if v war in Q, aber nicht jetzt then
                  füge v am Kopf von Q ein
            end
   end;
```

Algorithmus von Dijkstra

Die Datenstrukturen entsprechen denjenigen des Algorithmus von Moore. Statt der Queue Q wird hier ein boolesches Feld SET mit

$$SET[v] = \begin{cases} 0, \text{ falls v nicht in der Menge der markierten Knoten} \\ 1, \text{ falls v in der Menge der markierten Knoten} \end{cases}$$

benutzt. Voraussetzung für diesen Algorithmus ist $W[u,v] \geq 0$ für alle u und v.

Der Algorithmus beginnt an der Quelle und berechnet die direkten Wege zu allen erreichbaren Nachbarknoten. Da negative Kantenlängen ausgeschlossen sind, ist es offensichtlich, daß es zu dem von der Quelle am wenigsten weit entfernten Knoten keinen kürzeren Weg geben kann. Alle anderen Wege müßten über solche Knoten führen, die selbst schon weiter von der Quelle entfernt sind. Die Entfernung zu dem nahesten Knoten kann somit festgelegt werden und der Knoten wird markiert. Es ist sinnvoll, die Berechnung an

diesem Knoten fortzusetzen und die Wege von diesem Knoten aus zu allen noch nicht markierten Nachbarknoten zu berechnen. Dieser Schritt wird so lange wiederholt, bis alle Knoten markiert sind.
Die Zeitkomplexität von dem hier vorgestellten Algorithmus ist $O(N^2)$.

Der Algorithmus hat die folgende Gestalt:

```
for i := 1 to N do
   begin
      SET[i]  := 0;
      DIST[i] := W[SOURCE,i]
   end;
SET[SOURCE] := 1;
for j := 1 to N - 1 do
   begin
      wähle u mit DIST[u] = min {DIST[v] | SET[v] = 0};
      SET[u] := 1;
      for alle v mit SET[u] = 0 do
         begin
            dv := DIST[u] + W[u,v];
            if DIST[v] > dv then
               begin
                  DIST[v] := dv;
                  PRED[v] := u
               end
         end
   end;
```

4.1.1.1 Parallele Algorithmen nach Moore-Pape

Im folgenden werden zwei verschiedene Algorithmen vorgestellt, die nach dem sequentiellen Algorithmus von Moore-Pape entwickelt worden sind. Die **while**-Schleife wird von mehreren Prozessen parallel ausgeführt.

Der erste Algorithmus ist eine nicht symmetrische parallele Version, in der ein ausgezeichneter Prozeß die Verantwortung für die Synchronisierung zwischen den Prozessen trägt [DPL,80].
Nach der Initialisierung wird der Ausgangsknoten aus der Queue Q entnommen. Das Entfernen eines Knotens vom Kopf von Q wird von der Prozedur *get_u* bewerkstelligt. Dabei hat nur der Prozeß, der *get_u* aufgerufen hat, Zugriff auf Q. Die Prozedur *reachout* entspricht der inneren **for**-Schleife des sequentiellen Algorithmus. Es wird für alle Knoten v, die von u über eine Kante erreichbar sind, geprüft, ob für die Entfernung von der Quelle nach v ein kürzerer Weg über den Knoten u existiert. Falls dies der Fall ist, wird in DIST[v] der bis dahin kürzeste Weg von der Quelle nach v gespeichert und v durch die Prozedur *insert* in die Queue eingefügt.
Der Algorithmus braucht drei globale Variablen (WAIT, DONE und MSYN) für die Synchronisation zwischen den P Prozessen. Die Variable MSYN wird nur von dem ausgezeichneten Prozeß MAIN verändert. Der Prozeß MAIN hat bei der Entnahme von Elementen aus Q immer Vorrang gegenüber den anderen Prozessen. Hat MSYN den Wert "true", so müssen die untergeordneten Prozesse warten, bis der ausgezeichnete Prozeß ein Element aus Q entfernt hat. Dann erhält MSYN den Wert "false", und die übrigen Prozesse können mit dem Aufruf von *get_u* fortfahren.
Nachdem alle untergeordneten Prozesse die Initialisierung beendet haben und sich x der untergeordneten Prozesse in der **repeat**-Schleife befinden, müssen die restlichen untergeordneten Prozesse die Prozedur *reachout* ausführen. Falls einer der Prozesse ein Element in die Queue einfügt, wird dieses von dem ausgezeichneten Prozeß entnommen. Wenn die Queue leer bleibt und alle untergeordneten Prozesse die Prozedur *reachout* vollständig abgearbeitet haben, bleiben diese schließlich in der **repeat**-Schleife. MSYN hat dann den Wert "true", die Variable WAIT hat den Wert P-1 und der ausgezeichnete Prozeß setzt DONE auf den Wert 1. Darauf beenden alle anderen Prozesse, nach einer Erhöhung von DONE um 1, ihre Arbeit.

Nicht symmetrischer paralleler Algorithmus nach Moore-Pape

Prozeß MAIN:

```
    MSYN := true; WAIT := 0; DONE := 0;
    for i := 2 to P do
       create TASK(i);
    for u := 1 to N step P do
       DIST[u] := ∞;
    while WAIT < P - 1 do
       warten;
    DIST[SOURCE] := 0;
    call insert(SOURCE);
L1: call get_u(u);
    if u ≠ 0 then
       begin
          MSYN := false;
          call reachout(u);
          MSYN := true;
       end
    else
       if WAIT = P - 1 then
          begin
             DONE := 1;
             goto L2
          end;
    goto L1;
L2: while DONE < P do
       warten;
```

Prozeß TASK(i):

```
    for u := i to N step P do
       DIST[u] := ∞;
L1: while MSYN = true do
       begin
          increment WAIT;
          repeat
             if DONE > 0 then
                goto L2
          until MSYN = false;
          decrement WAIT;
       end;
    call get_u(u);
    if u > 0 then
       call reachout(u);
    goto L1;
L2: increment DONE;
```

Die folgenden Prozeduren werden auch für den zweiten Algorithmus benötigt.

```
    procedure get_u(u)
    begin
       lock Q;
       if Q ist nicht leer then
          entferne Knoten u am Kopf von Q
       else
          u := 0;
       unlock Q
    end;
```

```
procedure reachout(u)
begin
   for jede Kante (u,v) do
      begin
         newdv := DIST[u] + W[u,v];
         lock DIST[v];
         if DIST[v] ≤ newdv then
            unlock DIST[v]
         else
            begin
               PRED[v] := u;
               DIST[v] := newdv;
               unlock DIST[v];
               call insert(v)
            end
      end
end;


procedure insert(v)
begin
   lock Q;
   if v war niemals in Q then
      füge v am Ende von Q ein;
   if v war in Q, momentan jedoch nicht then
      füge v am Kopf von Q ein;
   unlock Q
end;
```

Der zweite Algorithmus ist eine symmetrische parallele Version, in der der ausgezeichnete Prozeß nur die P-1 untergeordneten Prozesse startet, sich dann aber genauso wie diese verhält [MaDe,82].
Die globale Variable INIT stellt sicher, daß das Feld DIST initialisiert ist, bevor die Entfernungen von der Quelle berechnet werden. Alle Prozesse rufen die Prozedur *get_u* auf. Die Prozesse, die ein Element aus Q entnommen haben, führen anschließend die Prozedur *reachout* aus. Ist Q für einen Prozeß P_i bereits leer, so setzt dieser eine lokale Variable DQEMPTY auf "true" und erhöht die globale Variable EMQ, die die Anzahl der Prozesse angibt, die Q leer gefunden haben. Ist durch einen anderen Prozeß in der Zwischenzeit ein Element in Q eingefügt worden, so kann P_i dieses Element entfernen und DQEMPTY wieder auf "false" setzen. Außerdem wird EMQ dekrementiert. Ist EMQ gleich P, so endet der Algorithmus.

Symmetrischer paralleler Algorithmus nach Moore-Pape

Prozeß MAIN:

```
INIT := 0; EMQ := 0;
initialisiere Q mit SOURCE;
for i := 1 to P - 1 do
   create TASK(i);
call TASK(P);
```

Prozeß TASK(i):

```
for u := i to N step P do
   DIST[u] := ∞;
DIST[SOURCE] := 0;
DQEMPTY := false;
increment INIT;
while INIT ≠ P do
   warten;
repeat
   call get_u(u);
   if u = 0 then
      if not DQEMPTY then
         begin
            increment EMQ;
            DQEMPTY := true
         end;
   else
      if DQEMPTY then
         begin
            decrement EMQ;
            DQEMPTY := false;
            call reachout(u)
         end
      else
         call reachout(u)
until EMQ = P;
```

Der Speedup der beiden vorgestellten parallelen Algorithmen ist durch den sequentiellen Zugriff auf die Queue beschränkt. In [Yoo,83] wird ein synchroner paralleler Algorithmus vorgestellt, in dem die Queue zwischen den P Prozessoren aufgeteilt wird. Dadurch können alle Prozessoren gleichzeitig Elemente in ihre Queue einfügen oder aus ihrer Queue entfernen. Durch diese Aufteilung wird der sequentielle Zugriff zur Queue umgangen. Da der Algorithmus in Stufen arbeitet, müssen die P Prozesse jedoch nach jeder Stufe synchronisiert werden. In Stufe i werden alle Knoten untersucht, die von der Menge S_i ausgehen. Gleichzeitig wird eine neue Menge S_{i+1} gebildet, die aus den Knoten besteht, deren Entfernung von der Quelle in Stufe i verringert worden ist. S_{i+1} ist also die Menge der Knoten, deren unmittelbare Vorgänger auf dem bisher kürzesten Weg von der Quelle in der Menge S_i enthalten sind. Am Beginn des Algorithmus besteht die Menge S_i nur aus der Quelle.
Die Zeitkomplexität des Algorithmus beträgt O(M*N/P+N*P).

Weiterhin wird in [Yoo,83] ein asynchroner paralleler Algorithmus beschrieben, dem kein Stufenkonzept zugrunde liegt. Jeder Prozeß kann wie beim synchronen Algorithmus Elemente aus seiner eigenen Queue entfernen, aber er darf in jede der P Queues Knoten einfügen. In welche Queue der Knoten eingefügt werden soll hängt von der Knotennummer ab. Beim Entfernen eines Knotens aus der Queue ist dann keine Synchronisation notwendig, da nur ein Prozeß Zugriff auf die Queue hat. Dagegen müssen die Prozesse beim Einfügen eines Knotens in eine Queue synchronisiert werden.
Sind die zu untersuchenden Knoten während der Ausführung des Algorithmus auf alle Prozesse gleichmäßig verteilt, so erhält man eine Zeitkomplexität von O(M*N/P). Im schlimmsten Fall können die zu untersuchenden Knoten nur einem Prozeß zugewiesen werden, und der parallele Algorithmus muß dadurch sequentiell arbeiten. Dann erhält man nur eine Zeitkomplexität von O(M*N).

4.1.1.2 Paralleler Algorithmus nach Dijkstra

Der folgende Algorithmus ist eine Parallelisierung des Algorithmus von Dijkstra [MaDe,82]. Der ausgezeichnete Prozeß MAIN startet P-1 Prozesse, die ihm untergeordnet sind.
Die globale Variable INIT zeigt die Beendigung der Initialisierungen der Felder DIST und SET aller Prozesse an. Dann wird die Quelle markiert und von diesem Knoten aus muß nun der Knoten mit der kleinsten Entfernung bestimmt werden.
Jeder der P-1 gebildeten Prozesse schlägt einen Knoten vor, der als nächstes markiert werden soll. Von allen zu betrachtenden Knoten eines Prozesses haben diese Knoten jeweils die kürzeste Entfernung von der Quelle. Diese Knoten werden im Feld C gespeichert. Hat ein Prozeß keinen Knotenvorschlag, so wird der Dummy-Knoten 0 in C eingetragen. Der ausgezeichnete Prozeß wählt von allen vorgeschlagenen Knoten denjenigen aus, der tatsächlich die kürzeste Entfernung von der Quelle hat, und markiert ihn. Von diesem Knoten aus wird dann die Berechnung fortgesetzt.
Die Variable NEXTU dient zur Kommunikation zwischen allen Prozessen. Jeder der P-1 Prozesse macht einen Knotenvorschlag und erhöht NEXTU. Der ausgezeichnete Prozeß muß während dieser Berechnung warten. Wenn der nächste zu markierende Knoten von dem ausgezeichneten Prozeß ausgewählt worden ist, erhält NEXTU den Wert 0. Während der Auswahlprozedur müssen alle anderen Prozesse warten.
Wenn der Graph dünnbesetzt ist, ändert sich der Wert von DIST[i] nur selten, da dann meist $W[u,i] = \infty$. Trotzdem ist die asymptotische Zeitschranke des Algorithmus $O(N*(N/P+P))$, die unter der Annahme $P << N$ zu $O(N^2/P)$ vereinfacht werden kann.

Paralleler Algorithmus nach Dijkstra

Prozeß MAIN:

```
DIST[SOURCE] := 0;
SET[SOURCE] := 1;
u := SOURCE;
NEXTU := 0;
INIT := 0;
for i := 1 to P - 1 do
   create TASK(i);
while INIT ≠ P - 1 do
   warten;
while u ≠ 0 do
   begin
      while NEXTU ≠ P - 1 do
         warten;
      u := 0;
      d := ∞;
      for i := 1 to P - 1 do
         begin
            j := C[i];
            if j ≠ 0 then
               if DIST[j] < d then
                  begin
                     d := DIST[j];
                     u := j
                  end
         end;
      if u ≠ 0 then
         SET[u] := 1;
      NEXTU := 0
   end;
```

Prozeß TASK(i):

```
for k := i to N step P - 1 do
   begin
      DIST[k] := ∞;
      SET[k] := 0
   end;
increment INIT;
while INIT ≠ P - 1 do
   warten;
while u ≠ 0 do
   begin
      C[i] := 0;
      d := ∞;
      for v := i to N step P - 1 do
         begin
            if SET[v] = 0 then
               begin
                  dv := DIST[u] + W[u,v];
                  if DIST[v] > dv then
                     begin
                        DIST[v] := dv;
                        PRED[v] := u
                     end;
                  if d > DIST[v] then
                     begin
                        d := DIST[v];
                        C[i] := v
                     end
               end
         end;
      increment NEXTU;
      while NEXTU ≠ 0 do
         warten
   end;
```

4.1.2 Kürzeste Wege zwischen allen Knotenpaaren

Der zu diesem Problem beste bekannte sequentielle Algorithmus ist der Algorithmus von Warshall-Floyd [Floy,62]. Das Verfahren ist sehr einfach. Im ersten Schritt vergleicht man alle Wege, die von einem beliebigen Knoten i zu einem anderen Knoten j führen, mit dem Umweg über Knoten 1. Ist der Umweg kürzer, so wird der alte Weg durch den Umweg ersetzt. Im zweiten Schritt werden alle Umwege über den Knoten 2 gebildet, im k-ten Schritt über den Knoten k usw.
Bezeichnet man die Länge des kürzesten Weges von i nach j nach Bestimmung des Umweges über den k-ten Knoten mit $d_{ij}^{(k)}$, so wird im (k+1)-ten Schritt für alle $i \neq k+1$ und alle $j \neq k+1$ die folgende Operation ausgeführt:

$$d_{ij}^{(k+1)} = \min \{d_{ij}^{(k)}, d_{i,k+1}^{(k)} + d_{k+1,j}^{(k)}\}.$$

Dabei läuft der Index k von 0 bis N-1.
Bei der Initialisierung ist W die bewertete Adjazenzmatrix des Graphen. Die Diagonale enthält Nullen, während Knotenpaare, zwischen denen keine Kante existiert, die Bewertung ∞ haben. Zum Schluß enthält W[i,j] den kürzesten Weg von i nach j, denn nach der k-ten Iteration ist W[i,j] die kürzeste Entfernung von i nach j mit Zwischenknoten aus der Menge {1, ..., k}. Die Zeitkomplexität des Algorithmus ist $O(N^3)$.

Algorithmus von Warshall-Floyd

```
for k := 1 to N do
   for i := 1 to N do
      if W[i,k] < ∞ and i ≠ k then
         for j := 1 to N do
            if W[i,j] > W[i,k] + W[k,j] then
               W[i,j] := W[i,k] + W[k,j];
```

4.1.2.1 Paralleler Algorithmus nach Warshall-Floyd

Der folgende parallele Algorithmus ist eine Parallelisierung des Algorithmus von Warshall-Floyd [DPL,80]. Der Algorithmus ist symmetrisch, da der ausgezeichnete Prozeß MAIN P-1 Prozesse startet und sich dann selbst wie diese verhält.
Jeder der P Prozesse berechnet in der k-ten Iteration für N^2/P Knoten die kürzeste Entfernung mit Zwischenknoten aus der Menge {1, ..., k}.
Die globale Variable SYN dient zur Synchronisation der P Prozesse. Die Prozesse müssen warten, bis alle die k-te Iteration beendet haben. Damit ist gewährleistet, daß in der k-ten Iteration W[i,j] die kürzeste Entfernung von i nach j mit Zwischenknoten aus der Menge {1, ..., k} enthält.
Die innerste **for**-Schleife muß nicht als kritischer Bereich behandelt werden. Es ist zwar möglich, daß der Wert von W[k,j] von einem Prozeß geändert wird, während sich ein anderer Prozeß zwischen der **if**-Abfrage und der Zuweisung befindet. Dies führt aber nicht zu falschen Ergebnissen, da W[k,j] nur verkleinert werden kann und damit die **if**-Abfrage auf jeden Fall erfüllt bleibt.
Die Zeitkomplexität beträgt $O(N^3/P+P*N)$. Wenn $P << N$ ist, kann P*N vernachläßigt werden, und man erhält $O(N^3/P)$.

Paralleler Algorithmus nach Warshall-Floyd

Prozeß MAIN:

```
SYN := 0;
for i := 1 to P - 1 do
   create TASK(i);
call TASK(P);
```

Prozeß TASK(ii):

```
for k := 1 to N do
   begin
      for i := ii to N step P do
         if W[i,k] < ∞ and i ≠ k then
            for j := 1 to N do
               if W[i,j] > W[i,k] + W[k,j] then
                  W[i,j] := W[i,k] + W[k,j];
      increment SYN;
      while SYN < P * k do
         warten
   end;
```

4.2 MINIMALE SPANNENDE BÄUME

Minimale spannende Bäume finden Anwendung in Versorgungs- bzw. Verteilungsnetzen und der Kabelverlegung. Für die Entwicklung paralleler Algorithmen zur Bestimmung eines minimalen spannenden Baumes (MST) eines bewerteten, zusammenhängenden, ungerichteten Graphen stehen drei klassische Algorithmen zur Verfügung:

1. Algorithmus von Prim-Dijkstra [Prim,57], [Dijk,59]
2. Algorithmus von Kruskal [Krus,56]
3. Algorithmus von Sollin [Sol,77]

Alle drei Algorithmen sind sogenannte **Greedy**-Algorithmen. Sie sehen nur einen Schritt (= Kante) voraus und machen das beste, was in diesem Schritt möglich ist.

Algorithmus von Prim-Dijkstra

In diesem Algorithmus wird mit einem Wald von N isolierten Knoten gestartet. Ein willkürlicher Knoten des Graphen wird als Startknoten ausgewählt. Dieser Knoten wird mit dem Knoten, zu dem er die kürzeste Entfernung hat verbunden. Diese beiden Knoten werden als Teilgraph betrachtet. Die Matrix W ist die bewertete Adjazenzmatrix. NEAR[i] gibt an, welcher Knoten von i die kürzeste Entfernung hat. Wenn der Knoten i schon im Teilgraph ist, hat NEAR[i] den Wert 0.
In den folgenden Schritten wird unter den isolierten Knoten der Knoten ausgewählt, der die kürzeste Entfernung zum Teilgraph hat, und zu diesem hinzugenommen. Der Algorithmus stoppt, wenn N-1 Knoten ausgewählt worden sind.
Der sequentielle Algorithmus hat die Zeitkomplexität $O(N^2)$ und das folgende Aussehen:

```
T := ∅;
NEAR[1] := 0; (* Anfangen mit Knoten 1 *)
for i := 2 to N do
   begin
      NEAR[i] := 1;
      DIST[i] := W[1,i]
   end;
while |T| < N - 1 do
   begin
      wähle next mit DIST[next] = min {DIST[i] | NEAR[i] ≠ 0};
      T := T ∪ {(next,NEAR[next])};
      NEAR[next] := 0;
      for j := 1 to N do
         if NEAR[j] > 0 and W[j,next] < W[j,NEAR[j]] then
            begin
               NEAR[j] := next;
               DIST[j] := W[j,next]
            end
   end;
```

Algorithmus von Kruskal

Zu Beginn des Algorithmus ist E die Menge aller Kanten im Graphen G = (V,E). Aus dieser Menge soll eine Kante mit minimalem Gewicht bestimmt werden, die dann aus E entfernt wird. In jedem folgenden Schritt wird von den übrigen Kanten des Graphen die kleinste Kante ausgewählt, so daß kein Kreis mit den schon vorher bestimmten Kanten entsteht. Wenn N-1 Kanten ausgewählt worden sind, bilden diese den minimalen spannenden Baum.
Eine Möglichkeit für die Verwaltung der Kanten in E ist eine sortierte sequentielle Liste. Es ist aber nicht nötig, alle Kanten zu sortieren, solange die nächste Kante auf einfache Weise bestimmt werden kann. Dies kann mit Hilfe der Datenstruktur Heap erreicht werden.
Ein **Heap** ist ein Binärbaum, bei dem sich das Niveau der Blätter höchstens um eins unterscheidet. Alle Söhne enthalten Elemente, die größer oder gleich als die Elemente ihrer Väter sind [AHU,74].
Eine geeignete Datenstruktur zur Speicherung eines Heap ist ein Array A, wobei A[1] die Wurzel enthält und in A[2i] und A[2i+1] die Elemente gespeichert sind, die linker und rechter Sohn (falls existent) des Knotens sind, der in A[i] gespeichert ist.
Für ein solches Array A mit den Elementen A[i], $1 \leq i \leq n$ gilt die folgende Heap-Eigenschaft:

$A[i] \leq A[2i]$ für $1 \leq i \leq n/2$ und
$A[i] \leq A[2i+1]$ für $1 \leq i \leq n/2$.

Die nächste auszuwählende Kante kann in O(log M) Zeit ermittelt werden. Für den Aufbau des Heap ist O(M) Zeit nötig. Folglich hat der gesamte Algorithmus die Zeitkomplexität O(M log M).

Der sequentielle Algorithmus hat die folgende Gestalt:

```
T := ∅;
while |T| < N - 1 do
   begin
      wähle kleinste Kante (u,v) aus der Menge E;
      if (u,v) erzeugt mit den Kanten in T keinen Kreis then
         T := T ∪ {(u,v)};
      streiche Kante (u,v) aus der Menge E
   end;
```

Algorithmus von Sollin

Unter allen Algorithmen, die zur Bestimmung eines minimalen spannenden Baumes eines Graphen bestehen, ist der Algorithmus von Sollin der am besten für die Entwicklung von parallelen Algorithmen geeignete, da der Algorithmus unter dem Gesichtspunkt paralleler Verarbeitung entwickelt worden ist.
Der Algorithmus von Sollin startet mit einem Wald von N isolierten Knoten, wobei jeder Knoten als ein Baum angesehen wird. In der for-Schleife wird für jeden Baum in dem Wald die kleinste Kante, die einen Knoten dieses Baumes mit einem Knoten eines anderen Baumes verbindet, bestimmt. Alle diese Kanten werden zu dem Wald hinzugenommen, mit der Einschränkung, daß zwei Bäume niemals durch mehr als eine Kante verbunden sein dürfen. (Verbindungen zwischen Knoten, durch die ein Kreis entsteht, werden willkürlich aufgelöst). Die Iterationen werden solange fortgeführt, bis es nur noch einen Baum im Wald gibt, den minimalen spannenden Baum. Bei der sequentiellen Implementierung sind höchstens $O(N^2)$ Vergleiche nötig, um die kleinste Kante, die mit jedem Knoten inzident ist, zu finden. Da die Anzahl der Bäume nach jeder for-Schleife um einen Faktor von zwei reduziert wird, sind höchstens log N Iterationen notwendig, um den minimalen spannenden Baum zu finden. Folglich hat der sequentielle Algorithmus die Komplexität $O(N^2 \log N)$.

Für einen gegebenen Graphen G = (V,E) kann der Algorithmus dann wie folgt beschrieben werden:

```
F0 := (V,∅); (* der Wald wird mit N isolierten Knoten initialisiert *)
i := 0;
while es ist mehr als ein Baum in Fi do
   begin
      for jeden Baum Tj im Wald Fi do
         wähle kleinste Kante (u,v), die Knoten u in Tj mit
         Knoten v in einem anderen Baum Tk im Wald Fi verbindet;
      diese Kanten, zu Fi hinzugenommen, bilden den Wald Fi+1;
      i := i + 1
   end;
```

4.2.1 Parallele Algorithmen nach Prim-Dijkstra

Im folgenden werden zwei verschiedene Algorithmen nach dem Algorithmus von Prim-Dijkstra vorgestellt.

Der erste Algorithmus ist eine nicht symmetrische parallele Version [DeYo,81].
Nach der Initialisierung startet der Prozeß MAIN P-1 Prozesse (TASK1). Während die P-1 untergeordneten Prozesse die Prozedur *shortest_distance* ausführen, muß der ausgezeichnete Prozeß warten. Jeder dieser P-1 Prozesse untersucht N/(P-1) Knoten. Mit Hilfe der globalen Variablen VMIN wird der Knoten bestimmt, der die kürzeste Entfernung zum Teilgraph hat. Danach werden die Prozesse durch die Variable SYN synchronisiert, und die gefundene Kante wird von dem ausgezeichneten Prozeß zum Teilgraph hinzugenommen. Der gefundene Knoten next wird markiert, d.h. NEAR[next] = 0 gesetzt. Danach startet der Prozeß MAIN wieder P-1 Prozesse (TASK2). Diese Prozesse rufen die Prozedur *update* auf, die für das Neusetzen der Felder NEAR und DIST zuständig ist, falls sich für die Knoten des Teilgraphen durch die Hinzunahme von Knoten next eine kürzere Entfernung als bisher ergibt.
Die Prozedur *shortest_distance* hat die Komplexität O(N/P+P) und die Prozedur *update* die Komplexität O(N/P). Da im Prozeß MAIN N-1 Iterationen durchgeführt werden, hat der gesamte Algorithmus unter Berücksichtigung der Synchronisationszeiten die Zeitkomplexität $O(N^2/P+N*P)$.

Nicht symmetrischer paralleler Algorithmus nach Prim-Dijkstra

Prozeß MAIN:

```
T := ∅;
NEAR[1] := 0; (* Anfangen mit Knoten 1 *)
for i := 2 to N do
   begin
      NEAR[i] := 1;
      DIST[i] := W[1,i]
   end;
while |T| < N - 1 do (* N - 1 Knoten im MST *)
   begin
      DIST[1] = ∞;
      VMIN := ∞;
      SYN := 0;
      for i := 1 to P - 1 do
         create TASK1(i);
      while SYN < P - 1 do
         warten;
      T := T ∪ {(next,NEAR[next])};
      NEAR[next] := 0;
      SYN := 0;
      for i := 1 to P - 1 do
         create TASK2(i);
      while SYN < P - 1 do
         warten
   end;
```

Prozeß TASK1(i):

```
call shortest_distance(i,P-1);
increment SYN;
```

Prozeß TASK2(i):

```
call update(i,P-1);
increment SYN;
```

Die folgenden Prozeduren werden auch in dem zweiten Algorithmus benötigt.

```
procedure shortest_distance(i,proc)
begin
   ii := 1;
   for j := i to N step proc do
      if NEAR[j] > 0 and DIST[j] < DIST[ii] then
         ii := j;
   lock VMIN; (* nur nötig für proc ≥ 2 *)
   if DIST[ii] < VMIN then
      begin
         next := ii;
         VMIN := DIST[ii]
      end;
   unlock VMIN (* nur nötig für proc ≥2 *)
end;
```

```
procedure update(i,proc)
begin
   for j := i to N step proc do
      begin
         if NEAR[j] > 0 and W[j,next] < W[j,NEAR[j]] then
            begin
               NEAR[j] := next;
               DIST[j] := W[j,next]
            end
      end
end;
```

Der zweite symmetrische Algorithmus [Yoo,83] verläuft ähnlich wie der nicht symmetrische, jedoch wird hier die Synchronisation nicht von einem ausgezeichneten Prozeß vorgenommen. Der Prozeß MAIN startet P-1 Prozesse und verhält sich dann selbst wie diese. Die Synchronisation erfolgt an denselben Stellen wie bei dem nicht symmetrischen Algorithmus.
Die Komplexität des Algorithmus ist ebenfalls $O(N^2/P+N*P)$.

Symmetrischer paralleler Algorithmus nach Prim-Dijkstra

Prozeß MAIN:

```
T := ∅;
NEAR[1] := 0; (* Anfangen mit Knoten 1 *)
SYN := 0;
for i := 1 to P - 1 do
   create TASK(i);
call TASK(P);
```

Prozeß TASK(i):

```
for j := i + 1 to N step P do
   begin
      NEAR[j] := 1;
      DIST[j] := W[1,j]
   end;
lock SYN;
SYN := SYN + 1;
if SYN = P then
   begin
      DIST[1] := ∞;
      VMIN := ∞;
      SYN := 0
   end;
unlock SYN;
while SYN > 0 do
   warten;
```

```
while |T| < N - 1 do (* N - 1 Kanten im MST *)
   begin
      call shortest_distance(j,P);
      lock SYN;
      SYN := SYN + 1;
      if SYN = P then
         begin
            T := T ∪ {(next,NEAR[next])};
            NEAR[next] := 0;
            SYN := 0
         end;
      unlock SYN;
      while SYN > 0 do
         warten;
      call update(j,P);
      lock SYN;
      SYN := SYN + 1;
      if SYN = P then
         begin
            VMIN := ∞;
            SYN := 0
         end;
      unlock SYN;
      while SYN > 0 do
          warten;
   end;
```

4.2.2 Paralleler Algorithmus nach Kruskal

Im Algorithmus von [DeYo,81] werden zwei Prozesse *Producer* und *Consumer* gebildet, die asynchron arbeiten. Eine zirkuläre Queue Q wird als Nachrichtenpuffer benutzt.
Der Producer benutzt als Datenstruktur einen Heap, in dem alle Kanten mit der Kantenbewertung w und den inzidenten Knoten u und v gespeichert sind.

Der Producer schickt das oberste Element des Heap, welches die nächstkleinste Kante enthält, zur Queue Q. Wenn Q voll ist, muß der Producer solange warten, bis der Consumer ein Element aus Q entnommen hat. Solange Q nicht voll ist, schickt der Producer jeweils die nächstkleinste Kante an das Ende der Queue.
Der Consumer entnimmt fortlaufend Elemente vom Anfang der Queue, solange diese nicht leer ist. Dann untersucht er, ob durch diese Kante ein Kreis entsteht. Ist dies nicht der Fall, so wird die Kante zum minimalen spannenden Baum hinzugenommen, indem die beiden Teilbäume, in denen die inzidenten Knoten enthalten sind, vereinigt werden.
Die Berichtigung des Heap kann wegen der bestehenden Präzedenzbedingung nicht parallel ausgeführt werden. Daher bleibt die Komplexität des Algorithmus O(M log M) [DeYo,81].

Paralleler Algorithmus nach Kruskal

Prozeß MAIN:

```
T := ∅;
call build_Heap;
create PRODUCER;
call CONSUMER;
```

Prozeß PRODUCER:

```
last := M;
rear := 0;
while CONSUMER lebt do
   begin
      while Q voll do
         warten;
      rear := (rear + 1) mod b;   (* b = |Buffer| *)
      schicke oberstes Element des Heap an das Ende von Q;
      setze letztes Element des Heap an die oberste Stelle;
      last := last - 1;
      call Heap(1,last)   (* berichtige Heap *)
   end;
```

Prozeß CONSUMER:

```
front := 0;
while |T| < N - 1 do
   begin
      while Q leer do
         warten;
      front := (front + 1) mod b;
      u := Q[front,1];   (* u, v sind die inzidenten Knoten
      v := Q[front,2];      der Kante mit Länge w *)
      w := Q[front,3];
      r1 := FIND(u);
      r2 := FIND(v);
      if r1 ≠ r2 then
         begin   (* (u,v) wird zu MST hinzugenommen *)
            T := T ∪ {(u,v)};
            call UNION(r1,r2)
         end
   end;
```

Die beiden Operationen FIND und UNION arbeiten auf Mengen. FIND(x) gibt als Wert die Teilmenge zurück, die x enthält. UNION(X,Y) bildet aus zwei disjunkten Teilmengen X und Y die Vereinigungsmenge.
Zu Beginn des Algorithmus bildet jeder Knoten eine Menge. Die Menge T, die zum Schluß die Kanten des minimalen spannenden Baumes enthält, wird als leere Menge initialisiert. Zu T wird die nächstkleinste Kante hinzugenommen, wenn dadurch kein Kreis entsteht. Der Test, ob durch eine bestimmte Kante ein Kreis mit den schon ausgewählten Kanten entsteht, wird mit Hilfe des Mengenvergleichs im CONSUMER realisiert.

Die beiden im Algorithmus verwendeten Prozeduren *build_Heap* und *Heap* sehen wie folgt aus [AHU,74]:

```
procedure build_Heap
begin
   for i := n/2 downto 1 do
      call Heap(i,n)
end;
```

```
procedure Heap(i,j)
begin
   if i ist nicht Blatt und ein Sohn von i enthält ein kleineres
      Element als i then
      begin
         sei k ein Sohn von i mit dem kleinsten Element;
         tausche A[i] und A[k];
         call Heap(k,j)
      end
end;
```

Wenn i > j/2, dann ist i ein Blatt, und in *Heap* ist die **if**-Abfrage nicht erfüllt. Daher kann die **for**-Schleife in *build_Heap* erst bei n/2 beginnen.

In [Yoo,83] wird gezeigt, wie mit Hilfe von *Software-Pipelining* die Heap-Operationen parallel ausgeführt werden können.

Eine Software-Pipeline ist eine geordnete Menge von Segmenten, in der der Output jedes Segmentes der Input des nachfolgenden Segmentes ist. Der Input des ersten Segmentes ist der Input des Algorithmus, der Output des letzten Segmentes ist der Output des Algorithmus.
Beim Software-Pipelining verhält sich jeder Prozeß wie ein Pipeline-Segment. Ein Element ist von Prozeß P_i verbraucht worden, wenn P_i seine Bearbeitung mit diesem Element beendet hat. Ein Element ist für P_i erzeugt worden, wenn das Element für die Verarbeitung durch P_i bereit ist.
Wenn nun ein Element von P_i verbraucht wird, dann erzeugt P_i ein Element für P_{i+1}. Wenn P_i ein Element erzeugt, dann verbraucht P_{i+1} dieses Element. Das Element, das vom ersten Prozeß verbraucht wird, ist der Input dieser Software-Pipeline. Das Element, das vom letzten Prozeß erzeugt wird, ist der Output der Software-Pipeline.
Das beschriebene Konzept wird in Abb. 10 dargestellt.

Mit dem Prinzip des Software-Pipelining kann ein Element aus einem M-elementigen Heap mit $P = \lceil \log M \rceil$ Prozessoren in konstanter Zeit entfernt werden. Die Zeitkomplexität des Algorithmus beträgt O(M) [Yoo,83].
Der Algorithmus benutzt ein Array zum Implementieren des Heap. Der Heap ist ein vollständiger Binärbaum, wobei evtl. Knoten mit dem Wert ∞ gefüllt werden. Jedem Niveau des Baumes wird ein Prozessor zugeordnet. Die Prozeßkommunikation geschieht über einen gemeinsamen Speicher. Während der Ausführung eines Algorithmus ist ein Knoten *voll*, wenn er einen Wert enthält (einschließlich ∞). Ein Knoten ist *leer*, wenn der Wert an seinen Vaterknoten weitergegeben und kein neuer Wert von den Söhnen empfangen worden ist. Der erste Prozeß wird der Wurzel zugeordnet. Er schickt das oberste Element, welches die nächstkleinste Kante enthält, zur Queue Q. Anschließend wartet er darauf, daß Prozeß 2 die Wurzel wieder füllt, um wieder die nächstkleinste Kante zu Q schicken zu können. Wenn Q voll ist, wartet Prozeß 1 darauf, daß der Consumer ein Element aus Q entnimmt.

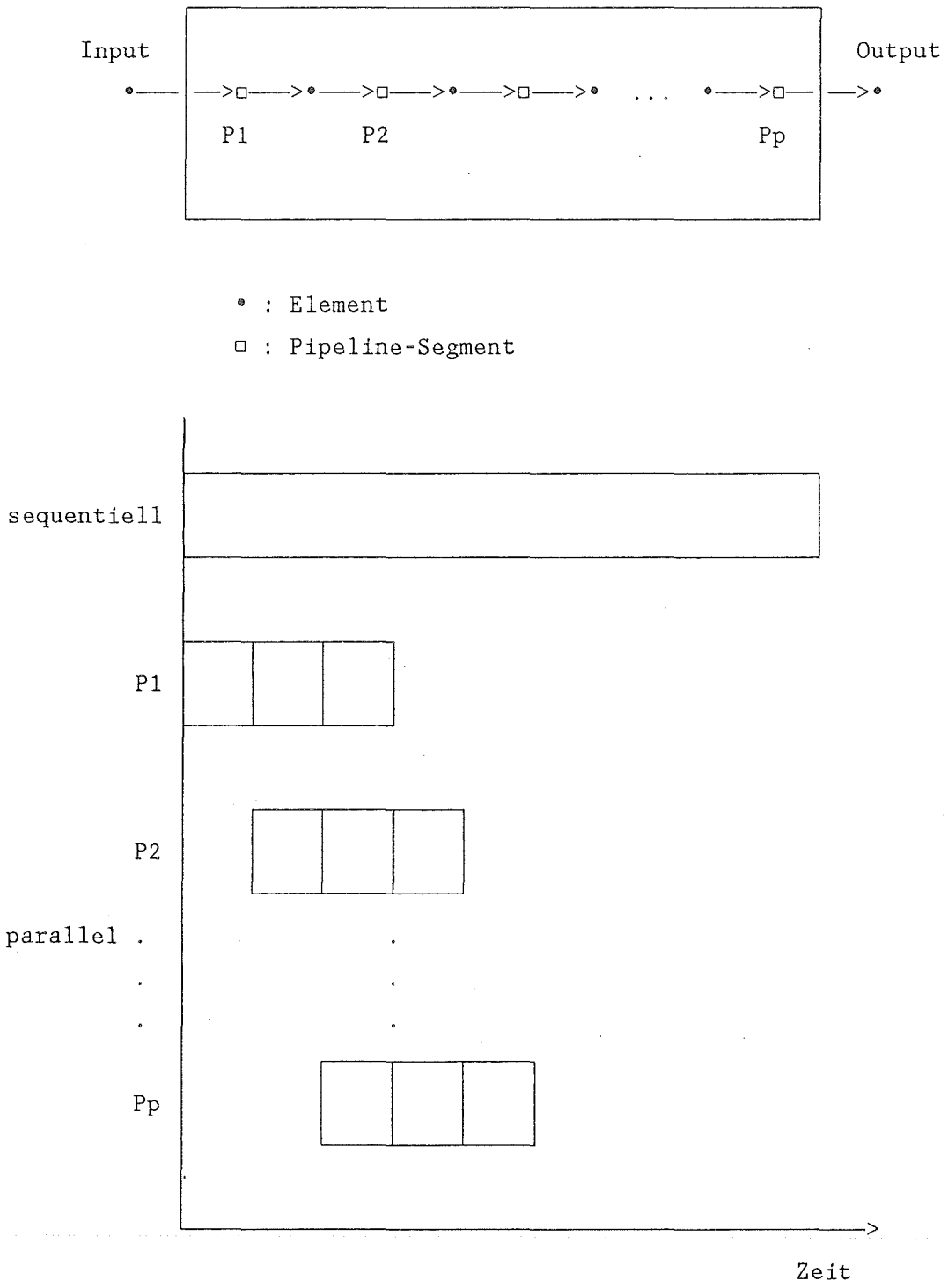
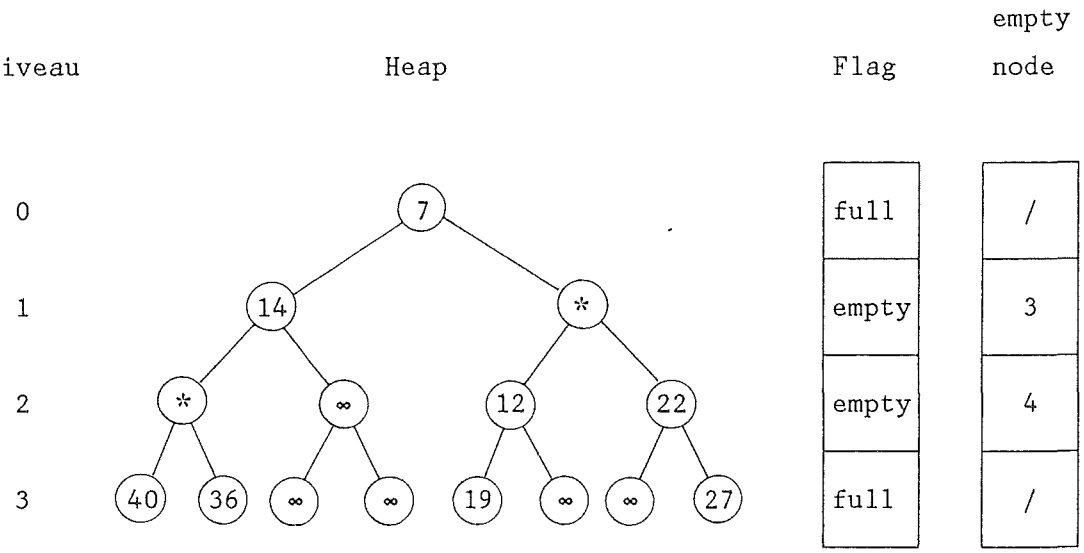


Abb. 10. Software Pipelining

Beispiel 4.1



Ein Array FLAG zeigt an, auf welchem Niveau ein leerer Knoten ist.

$$FLAG[i] = \begin{cases} \text{empty, falls Niveau } i \text{ einen leeren Knoten enthält} \\ \text{full, sonst} \end{cases}$$

Wenn FLAG[i] = empty ist, dann enthält empty_node[i] den Knoten, der leer ist.
Ist FLAG[i-1] = empty und FLAG[i] = full, dann wird der leere Knoten auf Niveau i-1 mit dem Wert des entsprechenden Sohnes gefüllt. Wird ein Blatt leer, so erhält es den Wert ∞. Die Prozedur stoppt, wenn die Wurzel den Wert ∞ enthält, d.h. der Baum ist leer.

Weiterhin wird in [Yoo,83] eine parallele Methode für die Initialisierung des Heap beschrieben. Sie entspricht im wesentlichen dem sequentiellen Algorithmus von [AHU,74], aber alle Knoten auf dem gleichen Niveau des Heap werden parallel verarbeitet.

4.2.3 Paralleler Algorithmus nach Sollin

Der folgende nicht symmetrische Algorithmus ist eine einfache parallele Umwandlung des Algorithmus von Sollin [DeYo,81].
Nach der Initialisierung werden von dem ausgezeichneten Prozeß MAIN P-1 Prozesse gebildet. Zu Beginn des Algorithmus bildet jeder Knoten einen Baum. In den untergeordneten Prozessen wird parallel für jeden Baum in dem Wald die kleinste Kante, die einen Knoten i dieses Baumes mit einem Knoten j eines anderen Baumes verbindet, bestimmt. Die inzidenten Knoten zu dieser Kante werden in den Arrays Vi und Vj gespeichert. Erst wenn alle P-1 Prozesse diese Ausführung beendet haben, kann der ausgezeichnete Prozeß prüfen, welche der bestimmten Kanten zu dem minimalen spannenden Baum hinzugenommen werden sollen. Dies geschieht mit Hilfe der Operationen FIND und UNION. Die zu einem Baum gehörenden Knoten sind in einer Menge enthalten. So kann leicht festgestellt werden, ob bestimmte Knoten in demselben Baum enthalten sind. Ist dies nicht der Fall, werden die beiden Bäume vereinigt. Besteht der Wald anschließend noch aus mehr als einem Baum, so werden erneut P-1 Prozesse gebildet, die die kleinsten Kanten zwischen den Bäumen bestimmen. Dies wird höchstens log N mal wiederholt, bis der Wald nur noch einen Baum enthält, den minimalen spannenden Baum.
Jeder Prozeß untersucht in jeder Iteration N/(P-1) Knoten und bestimmt für jeden Knoten denjenigen Knoten, der nicht im Baum ist und zu dem die Entfernung am kürzesten ist.
Die Komplexität für das Bestimmen der kleinsten Kante ist $O(N^2/P)$. Das Bilden der Prozesse in jeder Iteration braucht O(P) Zeit. Da der Algorithmus log N Iterationen benötigen kann, erhält man die Komplexität $O((N^2/P+P) \log N)$.

Paralleler Algorithmus nach Sollin

Prozeß MAIN:

```
T := ∅;
while |T| < N - 1 do
   begin
      SYN := 0;
      for i := 1 to N do
         VMIN[i] := ∞;
      for i := 1 to P - 1 do
          create TASK(i);
      while SYN < P do
         warten;
      for i := 1 to N do
         if VMIN[i] ≠ ∞ then
            begin
               r1 := FIND(Vi[i]);
               r2 := FIND(Vj[i]);
               if r1 ≠ r2 then
                  begin
                     T := T ∪ {(Vi[i],Vj[i])};
                     call UNION(r1,r2)
                  end
            end
   end;
```

Prozeß TASK(ii):

```
for i := ii to N step P - 1 do
   for j := 1 to N do
      begin
         r1 := FIND(i);
         r2 := FIND(j);
         if r1 ≠ r2 then
            begin
               lock VMIN[r1];
               if W[i,j] < VMIN[r1] then
                  begin
                     Vi[r1] := i;
                     Vj[r1] := j;
                     VMIN[r1] := W[i,j]
                  end;
               unlock VMIN[r1]
            end
      end;
increment SYN;
```

Ein paralleler Algorithmus nach Sollin mit einer anderen Aufteilung wird in [Yoo,83] beschrieben. Die Komplexität der beiden Algorithmen ist gleich.

4.3 TRAVELING SALESMAN PROBLEM

Das Traveling Salesman Problem (TSP) hat die Aufgabe, einen Zykel (Tour) minimaler Länge bzw. Kosten in einem bewerteten Graphen zu finden, wobei jeder Knoten genau einmal besucht wird. Bei diesem sehr einfach erscheinenden Problem ist der Rechenaufwand von exakten Lösungen sehr hoch. Er steigt mit wachsender Problemgröße exponentiell an. Daher sind gute heuristische Verfahren entwickelt worden, mit denen man auch bei großen Problemen zumindest eine nahezu optimale Lösung erhält.
Das TSP ist mit symmetrischer oder asymmetrischer Kostenmatrix und mit euklidischen oder nicht-euklidischen Entfernungen als Kosten formuliert worden. Im euklidischen TSP (ETSP) werden die Knoten auf Punkte im euklidischen Raum abgebildet und alle Entfernungen sind euklidische Entfernungen.
Zunächst soll eine exakte Lösung des asymmetrischen nicht-euklidischen TSP mit Hilfe eines Branch-and-Bound-Algorithmus bestimmt werden. Die Best-Bound-Suche minimiert die Anzahl der Teilprobleme, die untersucht werden müssen, obwohl der erforderliche Aufwand an Speicherplatz (d.h. die Tiefe des Suchbaumes) oft eine exponentielle Funktion der Problemgröße ist. Dieses exponentielle Anwachsen der Größe des Suchbaumes macht die Parallelisierung des Branch-and-Bound-Verfahrens so attraktiv: Es gibt genug Arbeit für die Prozessoren, und sie können unabhängig arbeiten.

4.3.1 Branch-and-Bound-Algorithmus

Ein Algorithmus der sich zur Parallelisierung eignet, ist der Algorithmus von Little, Murty, Sweeney und Karel [LMSK,63].

Der Algorithmus beginnt mit der Gesamtheit aller Lösungen, d.h. mit der Menge aller möglichen Touren, die im einzelnen unbekannt sind. Für diese Lösungsmenge wird eine untere Kostengrenze bestimmt. Diese Kostengrenze wird mit Sicherheit von keiner Lösung unterschritten.
Sie wird mittels Reduzierung der Kostenmatrix berechnet. Die *Kostenmatrix* eines Graphen läßt sich aus der bewerteten Adjazenzmatrix berechnen. Bei

Knotenpaaren, zwischen denen eine Kante existiert, wird der Wert aus der Adjazenzmatrix in die Kostenmatrix eingetragen. Die übrigen Kosten ergeben sich aus dem kürzesten Weg über die Zwischenknoten. Auf der Diagonalen wird ∞ eingetragen.

Eine Matrix heißt *reduzierte Matrix*, wenn sie in jeder Zeile und Spalte mindestens eine Null enthält. Die reduzierte Kostenmatrix erhält man durch Subtraktion geeigneter Konstanten von Zeilen und Spalten. Hat eine Tour die Kosten z(t) bei einer gegebenen Kostenmatrix und die Kosten $z_1(t)$ bei der zugehörigen reduzierten Kostenmatrix, dann gilt:

$$z(t) = L + z_1(t).$$

L ist die Summe der für die Reduktion notwendigen Konstanten. Wenn nur positive Kosten vorausgesetzt sind, dann ist L eine untere Schranke aller Touren der Originalmatrix.

Durch Reduzierung der Kostenmatrix erhält man i. allg. keine zusammenhängende Rundtour mit den Kosten L, sondern evtl. nur Kurzzyklen mit den Gesamtkosten L.

Nun wird die Lösungsmenge in zwei disjunkte Teilmengen aufgespalten. Lösung heißt hier eine Tour durch alle Knoten ohne Berücksichtigung der Kosten. Dabei ist eine Teilmenge immer dadurch gekennzeichnet, daß die in ihr zusammengefaßten Lösungen eine bestimmte Kante enthalten, während die Lösungen der anderen Teilmenge diese Kante nicht enthalten.

Diese Mengen werden als Knoten eines Zustandsraumbaumes dargestellt, wobei die Menge aller möglichen Touren die Wurzel bildet. Dieser Zustandsraumbaum ist ein Binärbaum, bei dem eine Verzweigung nach links die Hinzunahme einer bestimmten Kante bedeutet, während eine Verzweigung nach rechts dem Ausschluß jener Kante entspricht.

Für jede Teilmenge wird eine untere Schranke für die Kosten der Tour bestimmt. Mit Hilfe dieser Schranken wird die weitere Unterteilung der Touren gesteuert, bis eine einzige Tour erhalten ist, deren Kosten kleiner oder gleich den unteren Schranken aller anderen Teilmengen ist, die durch die Blätter des aktuellen Zustandsraumbaumes repräsentiert werden.

Für die Suche nach dem Lösungsknoten werden zwei Heuristiken verwendet:

1. Knoten-Auswahl-Heuristik
2. Kanten-Auswahl-Heuristik

Die *Knoten-Auswahl-Heuristik* wählt aus allen Blättern des aktuellen Baumes das Blatt aus, dessen untere Schranke die kleinste ist. Mit der *Kanten-Auswahl-Heuristik* wird erstens angestrebt, daß die Kostenuntergrenze der ersten Teilmenge möglichst niedrig bleibt. Das wird erreicht durch Wahl einer Kante mit den reduzierten Kosten von Null. Zweitens ist es wünschenswert, daß die Untergrenze der zweiten Teilmenge möglichst hoch wird. Dabei wird das Ziel verfolgt, den Entscheidungsbaum möglichst schlank, d.h. möglichst wenig verzweigt zu halten. Das läßt sich dadurch erreichen, daß man eine solche Kante (i,j) auswählt, bei deren Sperrung die Kosten stark steigen. Den Mindestbetrag, um den die Kosten steigen werden, erhält man leicht aus der Summe des kleinsten Kostenelementes in Zeile i und des kleinsten Kostenelementes in Spalte j.
Von allen Blättern des aktuellen Baumes wird mit Hilfe der Knoten-Auswahl-Heuristik der Knoten bestimmt, der am ehesten zur Lösung führt. Von dem Problem, das durch diesen Knoten bezeichnet wird, wird mit der Kanten-Auswahl-Heuristik eine Kante bestimmt. Nun werden zwei Söhne gebildet, wobei der eine Sohn alle Touren beinhaltet in denen die ausgewählte Kante enthalten ist und der andere Sohn alle Touren beinhaltet, die diese Kante nicht enthalten.

Die Kostenmatrix für einen Sohn wird aus der Kostenmatrix des Vaterknotens hergeleitet.
Wenn die Kante (i,j) in der Menge der Touren enthalten sein soll, werden Zeile i und Spalte j der Matrix gestrichen, da von diesem Knoten an, die Kante (i,j) in der Tour enthalten ist. Das Matrixelement (j,i) wird auf ∞ gesetzt, da die Hinzunahme dieser Kante zu einer Schleife führen würde.
Soll die Kante (i,j) aus der Menge der Touren ausgeschlossen werden, so wird nur das Matrixelement (i,j) auf ∞ gesetzt.

Der Algorithmus kann folgendermaßen beschrieben werden:

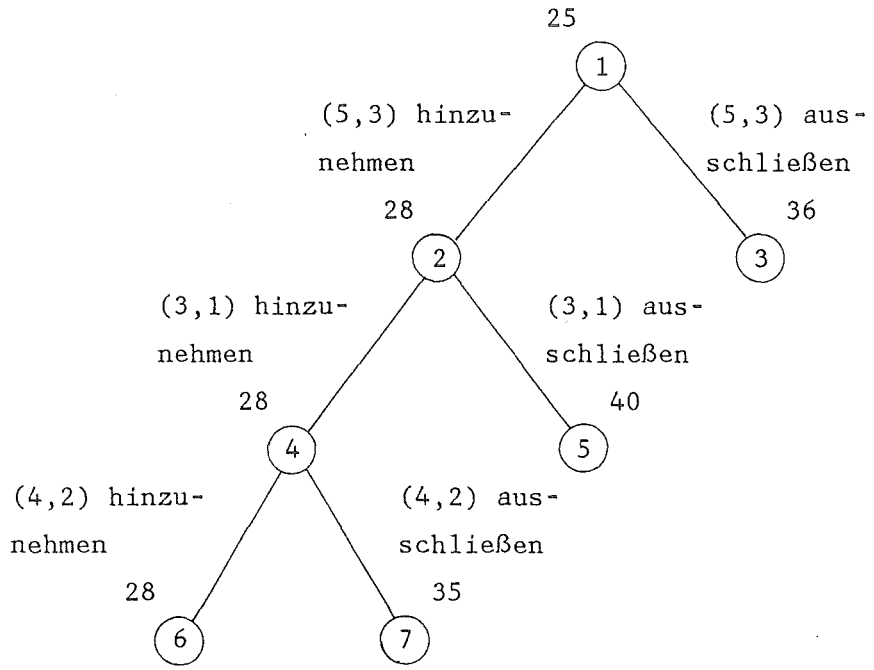
```
repeat
   wähle Knoten im Baum nach Knoten-Auswahl-Heuristik;
   wähle Kante (i,j) nach Kanten-Auswahl-Heuristik;
   for jeden Sohn der die Kante (i,j) einschließt/ausschließt do
      begin
         bilde Knoten und verbinde ihn mit dem Baum;
         leite Kostenmatrix für den Sohn her;
         reduziere Matrix und finde neue untere Schranke für alle
            Touren die durch diesen Sohn definiert werden
      end
until eine vollständige Tour mit Kosten kleiner oder gleich den
      unteren Schranken aller möglichen Touren erhalten ist;
```

Beispiel 4.2

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \qquad \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Kostenmatrix und eine zugehörige reduzierte Matrix mit L = 25.

Der zugehörige Zustandsraumbaum zu der Kostenmatrix sieht dann wie folgt aus:



Die optimale Rundreise ist dann 1 → 4 → 2 → 5 → 3 → 1 mit den Kosten 28.

4.3.1.1 Parallele Branch-and-Bound-Algorithmen

In [Moh,82] bzw. [Moh,83] werden zwei parallele Versionen nach dem Algorithmus von [LMSK,63] vorgestellt. In dem ersten Algorithmus wird die **for**-Schleife parallel ausgeführt, im zweiten Algorithmus die **repeat**-Schleife.
Als Datenstrukturen werden verschiedene N × N Kostenmatrizen verwendet, wobei N die Anzahl der Knoten im Graphen ist. Ein Zustandsraumbaum zeigt den aktuellen Zustand des Problems an. Jeder Prozeß hat eine lokale Datenstruktur, die die Kostenmatrix des Knotens enthält, der gerade von diesem Prozeß untersucht wird.

Der erste Algorithmus arbeitet synchron, da die Prozesse nach dem Bilden der Sohnknoten synchronisiert werden müssen. Ein ausgezeichneter Prozeß führt die ersten beiden Schritte des sequentiellen Algorithmus aus und übernimmt die Kontrolle für die **for**-Schleife. Das Bilden der Sohnknoten erfolgt parallel, indem jeder Prozeß einen Sohn bildet. In Kapitel 4.3.1

hat jeder Knoten nur zwei Söhne, wobei jeweils die ausgewählte Kante eingeschlossen oder ausgeschlossen wird. Damit erhält man nur eine Parallelität von zwei. Um einen höheren Parallelitätsgrad zu erhalten, kann der Algorithmus so modifiziert werden, daß mit mehrfacher Anwendung der Kanten-Auswahl-Heuristik mehrere Kanten ausgewählt werden. Dann können für N ausgewählte Kanten 2^N Söhne mit 2^N Prozessoren gebildet werden. Jeder Sohn entspricht einer möglichen Einschluß/Ausschluß-Kombination der ausgewählten Kanten. Da der ausgezeichnete Prozeß die Touren in disjunkte Mengen unterteilt, ist keine LOCK-Operation notwendig.

Im zweiten Algorithmus werden mehrere Prozesse gebildet, die asynchron den Baum der Teilprobleme untersuchen, bis eine Lösung gefunden ist. Die Auswahl des nächsten Knotens erfolgt auch hier nach der Knoten-Auswahl-Heuristik. Die Blätter sind hier jedoch im Gegensatz zu dem ersten Algorithmus in einer geordneten Liste gespeichert. Das erste Element der Liste ist der Knoten mit der kleinsten unteren Schranke. Jeder Prozeß entfernt wiederholt das nicht erforschte Teilproblem mit der kleinsten unteren Schranke aus der Liste, teilt das Problem (falls es nicht direkt lösbar ist) und fügt zwei neu gebildete Unterprobleme an die richtigen Stellen in der geordneten Liste der noch zu untersuchenden Teilprobleme ein. Für das Einfügen oder Entfernen von Elementen aus der Liste muß ein Prozeß die alleinige Kontrolle über die Liste haben. Diese Zeit ist aber relativ klein, verglichen mit der für die Problemteilung nötigen Zeit.
Hat ein Prozeß P einen Endknoten erreicht, der eine vollständige Tour darstellt und kleinere Kosten als alle anderen aktuellen Knoten hat, so wird eine Zählvariable inkrementiert. Sind alle Prozesse mit ihrer Arbeit fertig, so ist das Problem gelöst. Andernfalls muß P warten oder evtl. die Zählvariable dekrementieren und von einem anderen Knoten aus weitersuchen.
Da beide Algorithmen eine exponentielle Komplexität haben und eine große Prozessoranzahl erfordern, wird hier auf eine Implementierung verzichtet. Stattdessen werden zwei Näherungsverfahren für das euklidische TSP angegeben, die Heuristik des weitesten Einfügens und der Unterteilungsalgorithmus von Karp.

4.3.2 Heuristik des weitesten Einfügens

Die Heuristik des weitesten Einfügens [SDK,83] ist ein Näherungsalgorithmus zur Lösung des ETSP. Der Grundgedanke ist, eine Tour durch sukzessive Hinzunahme des am weitesten entfernten Knoten zu konstruieren. Der Ausgangsknoten (Quelle, Source) wird willkürlich gewählt. Dann wird der Knoten bestimmt, der die weiteste Entfernung von der Quelle hat und eine Tour von der Quelle zu diesem Knoten und zurück gebildet. Im weiteren Verlauf wird jeweils der Knoten ausgewählt, der von der teilweise gebildeten Tour die weiteste Entfernung hat und so zu der Tour hinzugenommen, daß die Tourkosten minimal sind. Dieser Schritt wird wiederholt, bis alle Knoten in der Tour enthalten sind.
Um die Suche nach dem Knoten, der die weiteste Entfernung von den Knoten der Teiltour V_T hat, effizient zu gestalten, wird ein Array DIST benutzt. Es enthält für jeden Knoten v, der nicht in der Teiltour ist, die Entfernung von v zu dem am nächsten liegenden Knoten aus V_T. Der Knoten f, der die weiteste Entfernung zu der Teiltour hat, ist der mit dem größten Wert im Array DIST. Dieser Knoten wird als nächster zu der Teiltour hinzugenommen. Nun muß bestimmt werden, wo der Knoten f in die Teiltour eingefügt werden soll. Dazu werden alle Kanten (i,j) in der aktuellen Teiltour, d.h. alle Kanten in der Menge E_T, untersucht, um die Kosten für das Einfügen von Knoten f zwischen den Knoten i und j zu bestimmen. Diese Kosten werden wie folgt mit Hilfe der Gewichtsmatrix W berechnet:

$$C[i,j] = W[i,f] + W[f,j] - W[i,j].$$

Es wird die Kante (t,h) ausgewählt, für die C[t,h] den kleinsten Wert hat. Der Knoten f wird dann zwischen den Knoten t und h eingefügt und die Länge der Tour berichtigt. Außerdem werden die Werte im Array DIST neu berechnet. Dazu wird das Minimum aus dem aktuellen Wert und der f-ten Zeile der Gewichtsmatrix W bestimmt.
Für symmetrische Graphen liegt die Länge der gefundenen Tour bis zu sieben Prozent über der Optimallösung [Quinn,83]. Die Zeitkomplexität des Algorithmus beträgt $O(N^2)$.
Der Algorithmus hat dann das folgende Aussehen:

```
V_T := {SOURCE};
E_T := {(SOURCE,SOURCE)};
W[SOURCE,SOURCE] := 0;
LENGTH := 0;
for alle Knoten u ∈ V - V_T do
   DIST[u] := W[SOURCE,u];
while |V_T| < N do
   begin
      wähle f mit DIST[f] = max {DIST[i] | i ∈ V - V_T};
      for jede Kante (i,j) ∈ E_T do
         C[i,j] := W[i,f] + W[f,j] - W[i,j];
      wähle (t,h) mit C[t,h] = min {C[i,j] | (i,j) ∈ E_T};
      E_T := E_T ∪ {(t,f),(f,h)} - {(t,h)};
      V_T := V_T ∪ {f};
      LENGTH := LENGTH + C[t,h];
      for alle Knoten u ∈ V - V_T do
         DIST[u] := min {DIST[u],W[f,u]}
   end;
```

4.3.2.1 Parallelisierung der Heuristik des weitesten Einfügens

Es gibt zwei Gründe, warum die Heuristik des weitesten Einfügens leicht zu parallelisieren ist:

1. Die Knoten können in beliebiger Reihenfolge durchsucht werden, auch parallel. (Dies gilt für beide Phasen der Iteration, sowohl beim Durchsuchen der Knoten, die nicht in der Tour sind, als auch beim Durchsuchen der Knoten, die in der Tour sind.)
2. In jeder Iteration wird genau ein Knoten zu der teilweise gebildeten Tour hinzugenommen.

Der ausgezeichnete Prozeß MAIN bildet P-1 Prozesse. Für die Synchronisation werden die Variablen INIT, COUNT1 und COUNT2 verwendet. Ein Array AUX der Länge N gibt an, welche Knoten bereits in der Tour sind und welche nicht. In der k-ten Iteration enthalten die ersten k Elemente des Arrays

die Namen der Knoten, die bereits in der Tour sind. Die Elemente k+1 bis N enthalten die Namen der Knoten, die nicht in der Tour sind. Beim Durchsuchen der Teilfelder untersucht jeder der P-1 Prozesse von Element i aus jedes (P-1)-te Element.
Die Prozedur *find_farthest_vertex* bestimmt den Knoten, der von der teilweise gebildeten Tour die weiteste Entfernung hat. Mit der Prozedur *determine_insertion_point* wird die Stelle bestimmt, an der der neue Knoten in die Tour eingefügt werden soll, so daß die Tourkosten minimal sind. Soll ein Knoten, dessen Name in AUX[j] gespeichert ist, als k-tes Element zu der teilweise gebildeten Tour hinzugenommen werden, so werden die Namen der Elemente AUX[j] und AUX[k] vertauscht. Dies geschieht in der Prozedur *add_vertex*.
Am Beginn des Algorithmus enthält AUX[i] den Namen von Knoten i, für alle $1 \leq i \leq N$.
Das erste Element, das in die Tour aufgenommen wird ist die Quelle.
Das Array TOUR enthält am Ende des Algorithmus die berechnete Tour, deren Länge in der Variablen LENGTH gespeichert ist. Die Variable chosen enthält den letzten Knoten, der zu der Tour hinzugenommen worden ist. Das Array CYCLE enthält die Knoten der Tour und das Array DIST die Entfernung der Knoten zu der Tour. gindex ist der Index im Array AUX des Knotens mit der weitesten Entfernung. GMAX ist die Entfernung des ausgewählten Knotens zu der Tour und GMIN enthält die minimalen Kosten, die entstehen wenn dieser Knoten zu der Tour hinzugenommen wird. inspt bezeichnet die Stelle an der der ausgewählte Knoten in die Tour eingefügt werden soll.

Die Zeitkomplexität der beiden Prozeduren *find_farthest_vertex* und *determine_insertion_point* beträgt jeweils O(N/P+P). Die Prozedur *add_vertex* hat die Komplexität O(1). Da für die Inititialisierung der Variablen O(N/P+P) Zeit nötig ist und die letzte **for**-Schleife im Prozeß MAIN O(N) Zeit benötigt, ergibt sich für den gesamten Algorithmus die Zeitkomplexität $O(N^2/P+N*P)$.

Der parallele Algorithmus [Quinn,83] kann dann wie folgt beschrieben werden:

Prozeß MAIN:

```
GMAX := -∞; GMIN := ∞; LENGTH := 0;
CYCLE[SOURCE] := SOURCE;
chosen := SOURCE;
for i := 1 to P - 1 do
   create TASK(i);
while INIT ≠ P - 1 do
   warten;
AUX[1] := SOURCE;
AUX[SOURCE] := 1;
COUNT1 := 0;
increment INIT;
for it_num := 1 to N - 1 do
   begin
      while COUNT1 ≠ P - 1 do
         warten;
      COUNT2 := 0;
      chosen := AUX[gindex];
      increment COUNT1;
      while COUNT2 ≠ P - 1 do
         warten;
      COUNT1 := 0;
      call add_vertex();
      increment COUNT2
   end;
index := SOURCE;
for i := 1 to N do
   begin
      TOUR[j] := index;
      index := CYCLE[index]
   end;
```

Prozeß TASK(i):

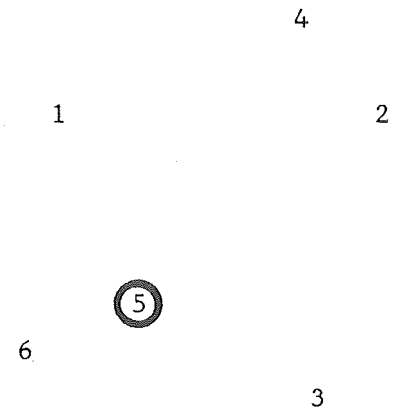
```
for j := i to N step P - 1 do
   begin
      DIST[j] := ∞;
      AUX[j] := j
   end;
increment INIT;
while INIT ≠ P do
   warten;
for it_num := 1 to N - 1 do
   begin
      call find_farthest_vertex(it_num,i);
      increment COUNT1;
      while COUNT1 ≠ P do
         warten;
      call determine_insertion_point(it_num,i,chosen);
      increment COUNT2;
      while COUNT2 ≠ P do
         warten
   end;


procedure find_farthest_vertex(it_num,i)
begin
   MAX := -∞;
   for j := it_num + i to N step P - 1 do
      begin
         vtxa := AUX[j];
         DIST[vtxa] := min {DIST[vtxa],W[vtxa,chosen]};
         if DIST[vtxa] > MAX then
            begin
               MAX := DIST[vtxa];
               index := j
            end
      end;
```

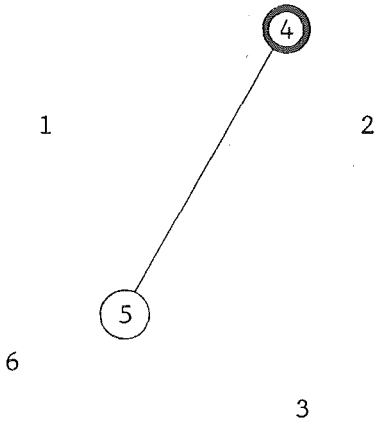
```
    lock GMAX;
    if MAX > GMAX then
       begin
          GMAX := MAX;
          gindex := index
       end;
    unlock GMAX
end;


procedure determine_insertion_point(it_num,i,chosen)
begin
    MIN := ∞;
    for j := i to it_num step P - 1 do
       begin
          vtxa := AUX[j];
          vtxb := CYCLE[vtxa];
          COST := W[vtxa,chosen] + W[chosen,vtxb] - W[vtxa,vtxb];
          if COST < MIN then
             begin
                MIN := COST;
                index := j
             end
       end;
    lock GMIN;
    if MIN < GMIN then
       begin
          GMIN := MIN;
          inspt := AUX[index]
       end;
    unlock GMIN
end;
```

```
procedure add_vertex()
begin
   LENGTH := LENGTH + GMIN;
   CYCLE[chosen] := CYCLE[inspt];
   CYCLE[inspt] := chosen;
   AUX[gindex] := AUX[it_num+1];
   AUX[it_num + 1] := chosen;
   GMAX := -∞;
   GMIN := ∞
end;
```

Beispiel 4.3

Gegeben sei der folgende Graph, in dem Knoten 5 als Quelle ausgewählt wird.





Knoten in der Tour

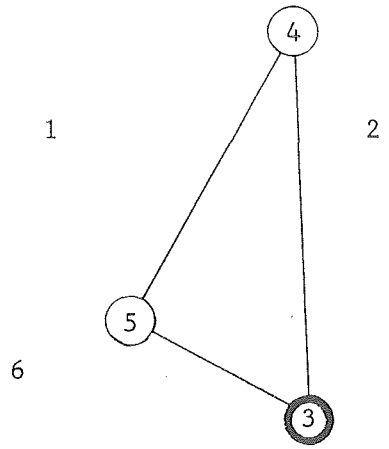
5	2	3	4	1	6

Knoten nicht in der Tour

Knoten in der Tour

5	4	3	2	1	6

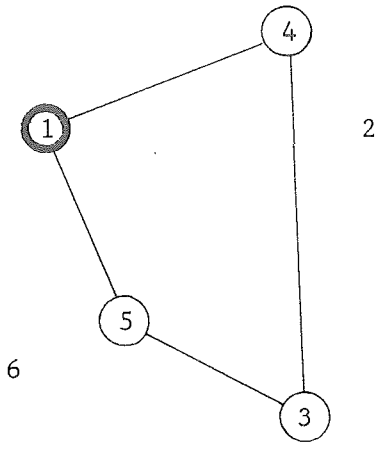
Knoten nicht in der Tour



Knoten in der Tour

5	4	3	2	1	6

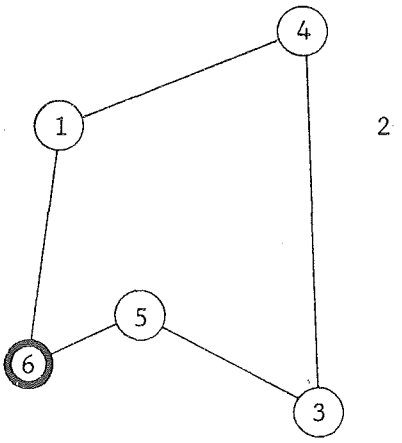
Knoten nicht in der Tour



Knoten in der Tour

5	4	3	1	2	6

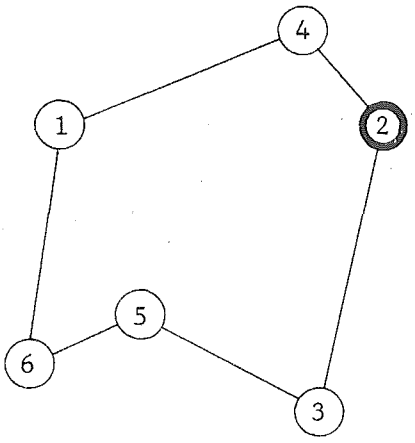
Knoten nicht in der Tour



Knoten in der Tour

5	4	3	1	6	2

Knoten nicht in der Tour



Knoten in der Tour

5	4	3	1	6	2

4.3.3 Unterteilungsalgorithmus von Karp

Dieser Näherungsalgorithmus ist eine rekursive Lösungsmethode des ETSP [Quinn,83]. Die Heuristik geht von der Annahme aus, daß jeder Nachfolgerknoten eines Knotens in der Tour nahe bei seinem Vorgänger liegt. Das ETSP wird "geographisch" in zwei Teilprobleme unterteilt, die unabhängig voneinander gelöst werden können. Die Vereinigung der beiden resultierenden Touren zu einer einzigen Tour bildet eine Näherungslösung des ETSP.
Eine Menge von Knoten, die sortiert nach x- (oder y-) Koordinaten ist, wird in zwei Teilmengen unterteilt. Der einzige Knoten, der in beiden Mengen enthalten ist, ist der Knoten mit dem mittleren x- (oder y-) Wert. Dieser Unterteilungsprozeß wird rekursiv weitergeführt, bis eine Menge entstanden ist, die klein genug ist. Dann ist eine Näherungslösung für das ETSP gefunden. Die Touren zweier Mengen werden vereinigt, indem zwei der mit dem gemeinsamen mittleren Knoten inzidenten Kanten entfernt werden. Die Tour wird dann mit einer Kante geschlossen (siehe Abb. 11).
Der Algorithmus hat die folgende Gestalt:

```
bilde Array X sortiert nach x-Koordinaten;
bilde Array Y sortiert nach y-Koordinaten;
bestimme Position der Elemente von X im Array Y und bilde Array XASSOC;
bestimme Position der Elemente von Y im Array X und bilde Array YASSOC;
call partition(1);


procedure partition(i)
begin
   if Anzahl der Elemente im Array von Partition i >
      maximale Unterteilungsgröße then
      begin
         if Verteilung der Knoten in x-Richtung ≥
            Verteilung in y-Richtung then
            unterteile Array X und Y nach x-Koordinaten
         else
            unterteile Array X und Y nach y-Koordinaten;
         for j := 2*i to 2*i+1 do
            call partition(j);
```

```
            kombiniere beide Touren aus Partition 2*i und Partition 2*i+1
        end
    else
        löse das Problem aus Partition i nach der Heuristik des
        weitesten Einfügens
end;
```

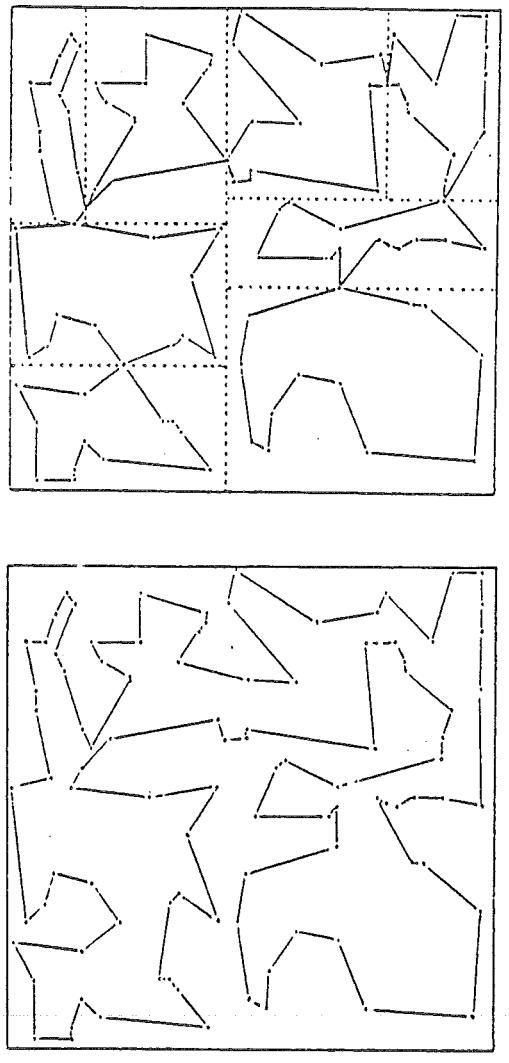


Abb. 11. Vereinigung der Teiltouren

4.3.3.1 Parallelisierung des Unterteilungsalgorithmus von Karp

Für die Parallelisierung des Unterteilungsalgorithmus von Karp wird eine Datenstruktur benötigt, mit der die Unterteilung parallel erfolgen kann. Unterteilt wird entweder nach den x-Koordinaten oder nach den y-Koordinaten, je nachdem in welche Richtung die Verteilung der Knoten größer ist. Um die Richtung festzulegen, müssen also die niedrigsten und die höchsten x- und y-Koordinaten der Menge der Knoten in jeder Partition bestimmt werden. Dies wird von der Prozedur *divide* geleistet. Ist die Richtung gefunden, so muß der mittlere Knoten bestimmt werden, so daß die Menge gleichmäßig in zwei Teilmengen aufgeteilt werden kann.
Ein Array X enthält die Knoten sortiert nach x-Koordinaten, entsprechend ein Array Y die Knoten sortiert nach y-Koordinaten. Zwei andere Arrays XASSOC und YASSOC dienen als Pointer zwischen X und Y. Die Position eines Knotens im Array Y, der in X[i] gespeichert ist, wird durch XASSOC[i] angegeben. Analog gilt X[YASSOC[i]] = Y[i]. Jede Partition hat ihre eigenen X, XASSOC, Y und YASSOC Arrays. Sie werden als Teilarrays eines großen Arrays implementiert auf die über einen Pointer zugegriffen wird. Die exakten x- und y-Koordinaten sind in den Arrays XLOC und YLOC gespeichert.

Angenommen, die Menge der N Knoten soll nach den x-Koordinaten parallel unterteilt werden. Jeder der P Prozessoren kopiert jeden P-ten Knoten aus der ersten Hälfte des Arrays in die erste Teilmenge des X-Arrays. Analog kopieren die P Prozessoren die zweite Hälfte des Arrays in das zweite Teilfeld. Dies geschieht in der Prozedur *setup*.
Das Kopieren der Elemente aus dem Array Y ist nicht so leicht, da die Position im neuen Teilfeld abhängig vom Index im Array X ist. Damit das Kopieren in die richtigen Teilfelder parallel ausgeführt werden kann, sind drei Schritte erforderlich.

1. Jedem Prozeß wird eine eigene Menge von N/P aufeinanderfolgenden Elementen in Y zugewiesen.
   Jeder Prozeß zählt die Anzahl der Elemente, die in eine der beiden Teilmengen gehören (*count_b_indices*).
   Dann müssen die Prozesse synchronisiert werden.
2. Dieser Schritt wird von einem einzigen Prozeß ausgeführt, der die Zähler prüft und jedem Prozeß Startpunkte für die beiden neuen Arrays zu-

weist, d.h. die Positionen in den zwei Arrays, wo sie mit dem Kopieren beginnen sollen.

3. Jeder Prozeß untersucht nochmal seine eigene Menge und schreibt jeden Knoten mit Hilfe der Startpunkte an die richtige Stelle im neuen Array (*copy_b_names*).

Das Programm arbeitet rekursiv, indem die Knotenmenge in immer kleinere Teilmengen unterteilt wird, bis die Anzahl der Elemente einer Partition i (ELS[i]) nicht mehr größer als ein vorgegebener Wert (MAX_ELS) ist. Die Quelle einer Partition i ist in SOURCE[i] gespeichert, der Index der Quelle in sindex[i].
Die kürzeste Tour einer Partition wird entsprechend Kapitel 4.3.2 nach dem Prinzip des weitesten Einfügens bestimmt (*find_farthest*, *find_insertion_point*). Wenn die Tour aus drei Knoten besteht, wird mit der Prozedur *insure_counterclockweise* geprüft, ob die Knoten gegen den Uhrzeigersinn durchlaufen werden. Ist das nicht der Fall, so wird die Reihenfolge der Knoten entsprechend geändert. Die gleiche Richtung aller Teiltouren ist wichtig für das Vereinigen der Touren, was in der Prozedur *combine* geschieht. Der Knoten, den beide Touren gemeinsam haben, ist jeweils die Quelle dieser Teiltouren. Das Arrayelement BEFORE[i] gibt den Vorgängerknoten, AFTER[i] den Nachfolgerknoten der Quelle in Partition i an.
Für die Vereinigung zweier Touren gibt es zwei Möglichkeiten. Die Kante in Partition i (Partition j), die zur Quelle führt und die Kante in Partition j (Partition i), die von der Quelle wegführt, werden gestrichen. Dann wird der Vorgängerknoten der Quelle in Partition i (Partition j) mit dem Nachfolgerknoten der Quelle in Partition j (Partition i) verbunden. Für beide Möglichkeiten werden die Kostenersparnisse (savings) berechnet und die Möglichkeit ausgewählt, durch die am meisten eingespart wird.
Ist der Algorithmus beendet, so enthält das Array BCYCLE die Vorgängerknoten und das Array CYCLE die Nachfolgerknoten jedes Knotens. Das Array TOUR enthält die Lösungstour mit der Länge length.

Der parallele Algorithmus sieht dann wie folgt aus:

Prozeß MAIN:

```
if MAX_ELS < 2 then
   length := -1.0
else
   begin
      ELS[1] := N;
      SOURCE[1] := X[1];
      sindex[1] := 1;
      piece_size := N;
      num_pieces := 1;
      while piece_size > MAX_ELS do
         begin
            piece_size := (piece_size + 2) div 2;
            num_pieces := 2 * num_pieces
         end;
      PARTITION_SIZE := num_pieces * piece_size;
      base := calc_base(1);
      partition(1,base);
      length := LEN[1];
      index := 1;
      for i := 1 to N do
         begin
            TOUR[i] := index;
            index := CYCLE[index]
         end
   end;


function calc_base(partition_number)
begin
   level := int(log(partition_number));
   first := 2 ** level;
   calc_base := PARTITION_SIZE * level + (partition_number - first) *
                (PARTITION_SIZE div first)
end;
```

```
function distance(i,j)
begin
   distance := sqrt((XLOC[i] - XLOC[j])**2 + (YLOC[i] - YLOC[j])**2)
end;


procedure partition(i,base)
begin
   if ELS[i] > MAX_ELS then
      begin
         divide(i,base);
         partition(2*i,base1);
         partition(2*i+1,base2);
         combine(i)
      end
   else
      solve_using_farthest_insertion(i,base)
end;


procedure divide(i,base)
begin
   if (XLOC[X[base+ELS[i]]] - XLOC[X[base+1]]) ≥
      (YLOC[Y[base+ELS[i]]] - YLOC[Y[base+1]]) then
      setup(i,base,X,XASSOC,Y,YASSOC,vertical)
   else
      setup(i,base,Y,YASSOC,X,XASSOC,horizontal)
end;
```

```
procedure setup(i,base,A,AASSOC,B,BASSOC,dividing_line)
begin
   part1 := 2 * i;
   part2 := 2 * i + 1;
   base1 := calc_base(part1);
   base2 := calc_base(part2);
   median := (ELS[i] + 1) div 2;
   ANZTASKS := max (P div 2**(i-1),1)
   for j := 1 to ANZTASKS - 1 do
      create TASK(j);
   for k := ANZTASKS to median step ANZTASKS do
      A[base1+k] := A[base+1];
   for k := median + ANZTASKS - 1 to ELS[i] step ANZTASKS do
      A[base2+k+1-median] := A[base+k];
   ELS[part1] := median;
   ELS[part2] := ELS[i] + 1 - median;
   increment SYN1;
   while SYN1 < ANZTASKS do
      warten;
   if ANZTASKS > 1 then
      count_b_indices(ANZTASKS);
   increment SYN2;
   while SYN2 < ANZTASKS do
      warten;
   lbase[1] := 0;
   hbase[1] := 0;
   for j := 2 to ANZTASKS do
      begin
         lbase[j] := lbase[j-1] + lindex[j-1];
         hbase[j] := hbase[j-1] + hindex[j-1]
      end;
   copy_b_names(ANZTASKS);
   while SYN3 < ANZTASKS - 1 do
       warten;
   SOURCE[part1] := A[base1+median];
   SOURCE[part2] := A[base2+1];
```

```
      if dividing_line = vertical then
         begin
            sindex[part1] := median;
            sindex[part2] := 1
         end
      else
         begin
            sindex[part1] := AASSOC[base1+median];
            sindex[part2] := AASSOC[base2+1]
         end
   end;
```

Prozeß TASK(j):

```
   for k := j to median step ANZTASKS do
      A[base1+k] := A[base+k];
   for k := median + j - 1 to ELS[i] step ANZTASKS do
      A[base2+k+1-median] := A[base+k];
   increment SYN1;
   while SYN1 < ANZTASKS do
      warten;
   count_b_indices(j);
   increment SYN2;
   while SYN2 < ANZTASKS do
      warten;
   copy_b_names(j);
   increment SYN3;
```

```
procedure count_b_indices(j)
begin
   lindex[j] := 0;
   hindex[j] := 0;
   for k := ((j-1)*ELS[i] div ANZTASKS) + 1 to j*ELS[i] div ANZTASKS do
      begin
         if BASSOC[base+k] ≤ median then
            lindex[j] := lindex[j] + 1;
         if BASSOC[base+k] ≥ median then
            hindex[j] := hindex[j] + 1
      end
end;


procedure copy_b_names(j)
begin
   for k := ((j-1)*ELS[i] div ANZTASKS) + 1 to j*ELS[i] div ANZTASKS do
      begin
         if BASSOC[base+k] ≤ median then
            begin
               lbase[j] := lbase[j] + 1;
               B[base1+lbase[j]] := B[base+k];
               BASSOC[base1+lbase[j]] := BASSOC[base+k];
               AASSOC[base1+BASSOC[base1+lbase[j]]] := lbase[j]
            end;
         if BASSOC[base+k] ≥ median then
            begin
               hbase[j] := hbase[j] + 1;
               B[base2+hbase[j]] := B[base+k];
               if ELS[part1] = ELS[part2] then
                  index = BASSOC[base+k]
               else
                  index = BASSOC[base+k] + 1;
               BASSOC[base2+hbase[j]] := (index mod ELS[part1]) + 1;
               AASSOC[base2+BASSOC[base2+hbase[j]]] := hbase[j]
            end
      end
end;
```

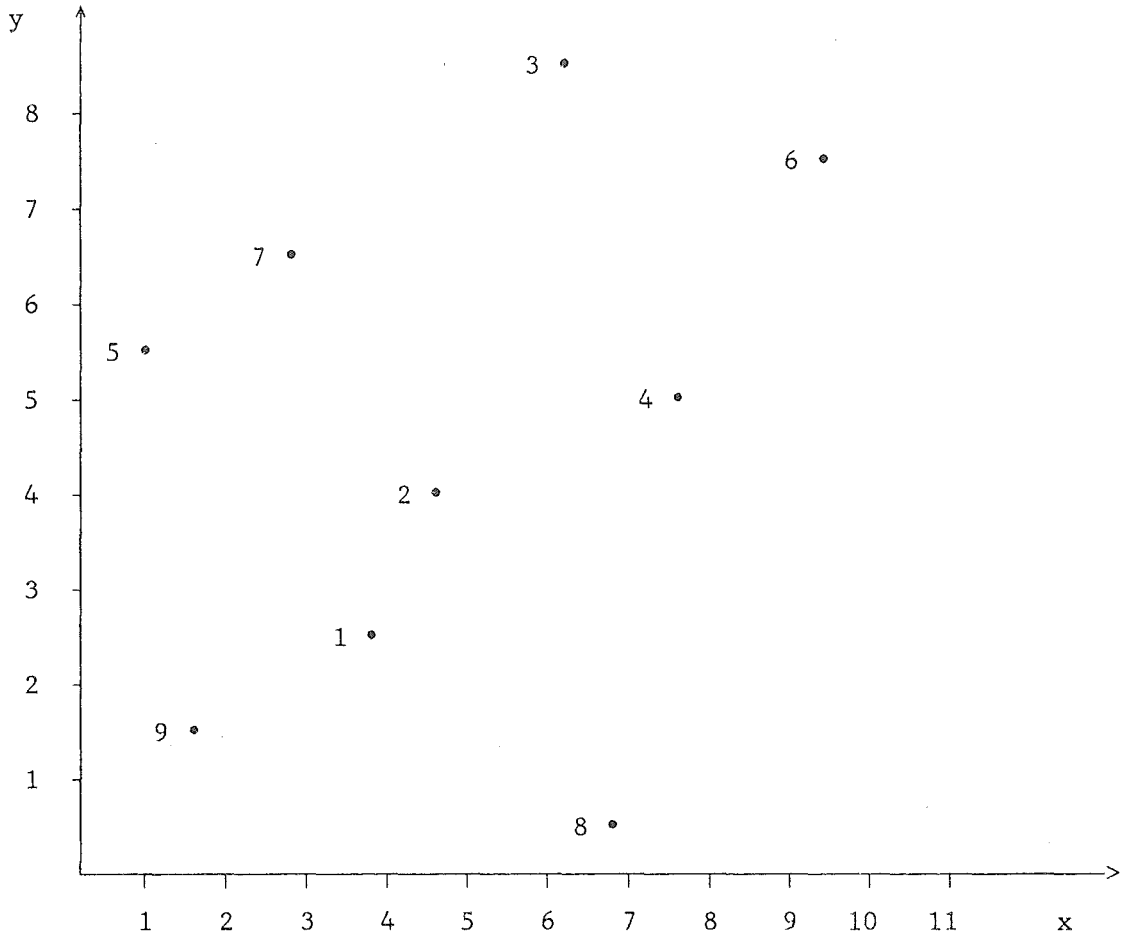
```
procedure combine(i)
begin
   part1 := 2 * i;
   part2 := 2 * i + 1;
   s := SOURCE[part1];
   savings1 := distance(BEFORE[part1],s) + distance(s,AFTER[part2])
               - distance(BEFORE[part1],AFTER[part2]);
   savings2 := distance(BEFORE[part2],s) + distance(s,AFTER[part1])
               - distance(BEFORE[part2],AFTER[part1]);
   if savings1 > savings2 then
      begin
         CYCLE[BEFORE[part1]] := AFTER[part2];
         BCYCLE[AFTER[part2]] := BEFORE[part1];
         CYCLE[s] := AFTER[part1];
         BCYCLE[s] := BEFORE[part2];
         LEN[i] := LEN[part1] + LEN[part2] - savings1
      end
   else
      begin
         CYCLE[BEFORE[part2]] := AFTER[part1];
         BCYCLE[AFTER[part1]] := BEFORE[part2];
         CYCLE[s] := AFTER[part2];
         BCYCLE[s] := BEFORE[part1];
         LEN[i] := LEN[part1] + LEN[part2] - savings2
      end;
      BEFORE[i] = BCYCLE[SOURCE[i]];
      AFTER[i] = CYCLE[SOURCE[i]]
end;
```

```
procedure solve_using_farthest_insertion(i,base)
begin
   LEN[i] := 0.0;
   for j := 2 to ELS[i] do
      for k := 1 to j - 1 do
         begin
            W[j,k] := distance(X[base+j],X[base+k]);
            W[k,j] := W[j,k]
         end;
   for j := 1 to ELS[i] do
      begin
         W[j,j] := 0.0;
         DIST[j] := ∞;
         AUX[j] := j
      end;
   AUX[1] := sindex[i];
   AUX[sindex[i]] := 1;
   chosen := sindex[i];
   TCYCLE[sindex[i]] := sindex[i];
   for it_num := 1 to ELS[i] - 1 do
      begin
         find_farthest(MAX,index);
         chosen := AUX[index];
         find_insertion_point(MIN,END1,END2);
         if it_num = 2 then
            insure_counterclockwise(i,base,END1,END2);
         TCYCLE[chosen] := END2;
         TCYCLE[END1] := chosen;
         AUX[index] := AUX[it_num+1];
         AUX[it_num+1] := chosen;
         LEN[i] := LEN[i] + MIN
      end;
   for j := 1 to ELS[i] do
      begin
         CYCLE[X[base+j]] := X[base+TCYCLE[j]];
         BCYCLE[X[base+TCYCLE[j]]] := X[base+j]
      end;
```

```
   BEFORE[i] := BCYCLE[X[base+sindex[i]]];
   AFTER[i] := CYCLE[X[base+sindex[i]]]
end;


procedure find_farthest(MAX,index)
begin
   MAX := -∞;
   for j := it_num + 1 to ELS[i] do
      begin
         vtxa := AUX[j];
         DIST[vtxa] := min {DIST[vtxa],W[vtxa,chosen]};
         if DIST[vtxa] > MAX then
            begin
               MAX := DIST[vtxa];
               index := j
            end
      end;
end;


procedure find_insertion_point(MIN,END1,END2)
begin
   MIN := ∞;
   for j := 1 to it_num do
      begin
         vtxa := AUX[j];
         vtxb := TCYCLE[vtxa];
         NEW_COST := W[vtxa,chosen] + W[chosen,vtxb] - W[vtxa,vtxb];
         if NEW_COST < MIN then
            begin
               MIN := NEW_COST;
               END1 := vtxa;
               END2 := vtxb
            end
      end;
end;
```

```
procedure insure_counterclockwise(i,base,END1,END2)
begin
   gend1 := X[base+END1];
   gend2 := X[base+END2];
   gfar := X[base+chosen];
   if XLOC[gend2] ≠ XLOC[gend1] then
      begin
         m := (YLOC[gend2] - YLOC[gend1]) / (XLOC[gend2] - XLOC[gend1]);
         b := YLOC[gend1] - m * XLOC[gend1];
         if YLOC[gfar] > (m * XLOC[gfar] + b) then
            above := true
         else
            above := false;
         if XLOC[gend1] < XLOC[gend2] then
            right := true
         else
            right := false
      end
   else
      begin
         if YLOC[gend1] < YLOC[gend2] then
            above := true
         else
            above := false;
         if XLOC[gend2] < XLOC[gfar] then
            right := true
         else
            right := false
      end;
   if (right and above) or ((not right) and (not above)) then
      begin
         temp := END1;
         END1 := END2;
         END2 := temp
      end
end;
```

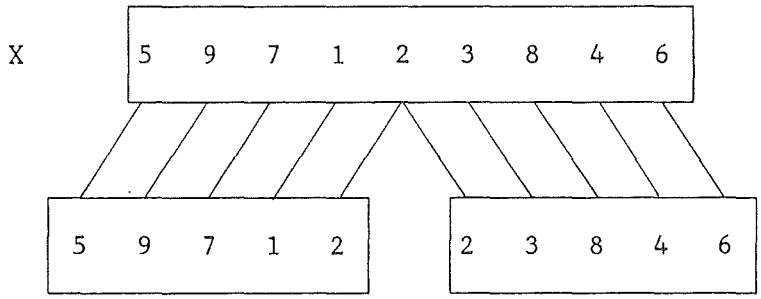
Beispiel 4.4



Für die obige Knotenverteilung ergeben sich die Arrays X und Y wie folgt:

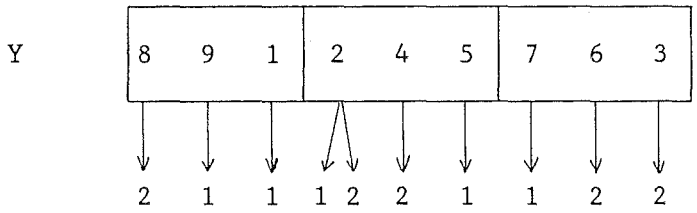
	1	2	3	4	5	6	7	8	9
X	5	9	7	1	2	3	8	4	6
XASSOC	6	2	7	3	4	9	1	5	8
YASSOC	7	2	4	5	8	1	3	9	6
Y	8	9	1	2	4	5	7	6	3

Es soll nun das erste Aufteilen der Arrays X und Y veranschaulicht werden. Da die Verteilung der Knoten in x-Richtung größer ist als in y-Richtung, wird zunächst das Array X in zwei Teilfelder unterteilt.



Für die Verteilung der Elemente aus Array Y auf zwei Teilfelder, die der Verteilung von Array X entsprechen, müssen die oben beschriebenen Schritte ausgeführt werden.

1. Jeder Prozeß bekommt eine Menge von N/P = 3 aufeinanderfolgenden Elementen in Y zugewiesen und bestimmt, wieviele Elemente in die erste und wieviele Elemente in die zweite Teilmenge gehören.

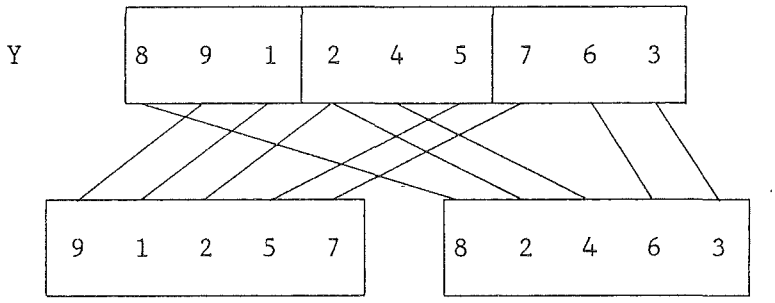


Teilmenge 1:	2	2	1
Teilmenge 2:	1	2	2

2. Die Zähler werden geprüft und Startpunkte für die beiden neuen Arrays zugewiesen.

Teilmenge 1:	0	2	4
Teilmenge 2:	0	1	3

3. Kopieren der Elemente in die richtigen Teilfelder.



5.0 IMPLEMENTIERUNG AUF DER CRAY X-MP/22

Einige der in Kapitel 4 vorgestellten Algorithmen sind auf der CRAY X-MP/22 in FORTRAN implementiert worden. Es ist untersucht worden, wie sich die Ausführungszeit der sequentiellen Programme gegenüber den parallelen Programmen mit Multitasking verhält. In beiden Fällen wurde die Vektorisierung ausgenutzt. Bei den verschiedenen Algorithmen ergeben sich je nach Kommunikations- und Synchronisationsaufwand unterschiedliche Beschleunigungen der Ausführungszeiten der sequentiellen Programme. Die Ausführungszeiten sind in Sekunden angegeben. Alle Programme bei denen der Code für beide Tasks getrennt vorliegt wurden ohne die Compiler-Option ALLOC=STACK übersetzt.

5.1 KÜRZESTE WEGE

Bei den Kürzeste-Wege-Algorithmen sind sowohl der Algorithmus von Moore-Pape und der Algorithmus von Dijkstra für die Bestimmung des kürzesten Weges von einem ausgezeichneten Knoten zu allen Knoten in einem Graphen, als auch der Algorithmus von Warshall-Floyd für die Bestimmung der kürzesten Wege zwischen allen Knotenpaaren implementiert worden.
Als Eingabe erhalten die Programme die Adjazenzmatrix eines Graphen. Bei der Adjazenzmatrix wurde die Anzahl der Kanten und die Verteilung der Kantenbewertungen in der Matrix variiert. Weiterhin ist das Verhalten der Programme bei unterschiedlicher Knotenzahl getestet worden. Die Matrizen sind durch positive Zufallszahlen erstellt worden. Bei den vollbesetzten Adjazenzmatrizen sind Zufallszahlen zwischen 1 und 100 erzeugt worden, bei allen anderen Adjazenzmatrizen Zahlen zwischen 1 und 10. Als Quelle wurde der Knoten 1 gewählt.
Bei den Lösungen der Algorithmen von Moore-Pape und Dijkstra können die kürzesten Wege bei gleicher Länge unterschiedlich sein, da die Vorgehensweise der Algorithmen verschieden ist. Außerdem können sich die kürzesten Wege der sequentiellen Programme von den kürzesten Wegen der parallelen

Programme bei gleicher Länge unterscheiden, da durch die parallele Verarbeitung eine andere Reihenfolge beim Durchlaufen der Knoten entsteht.

Algorithmus von Moore-Pape

Der parallele Algorithmus von Moore-Pape ist als symmetrischer Algorithmus, wie in Kapitel 4.1.1.1 beschrieben, implementiert worden.
In Tabelle 1 werden die Ausführungszeiten bei einer 200 × 200 Adjazenzmatrix mit variabler Kantenanzahl dargestellt. Es wird ein Speedup von 1.6 erreicht, der aber ab einer bestimmten Kantenanzahl wieder sinkt (siehe Abb. 12). Dies hat folgende Ursache:
Da jede Task den alleinigen Zugriff auf die Queue Q benötigt, müssen diese Anweisungen in einem kritischen Bereich stehen. Da die Länge von Q mit wachsender Kantenzahl zunimmt, wird der Overhead ab einer bestimmten Kantenanzahl zu groß, so daß der Speedup kleiner wird.
Dieser Sachverhalt wird durch die Ergebnisse in Tabelle 2 noch stärker verdeutlicht. Man erkennt in Tabelle 2 ein Ansteigen der Ausführungszeit bei steigender Knotenzahl und einer dichtbesetzten Adjazenzmatrix, sowohl im sequentiellen als auch im parallelen Programm. Jedoch ist die parallele Ausführungszeit entscheidend langsamer. Der parallele Algorithmus von Moore-Pape ist also offensichtlich für dichtbesetzte Adjazenzmatrizen nicht brauchbar.

Tabelle 1. Bestimmung des kürzesten Weges von der Quelle zu allen Knoten nach dem Algorithmus von Moore-Pape; Knotenanzahl N fest, Kantenanzahl M variabel

N = 200 M	ohne Multitasking	mit Multitasking	Speedup
100	0.00115	0.00145	0.79
200	0.00245	0.00217	1.13
400	0.02986	0.01905	1.57
600	0.04835	0.03031	1.60
800	0.04198	0.02808	1.50
1000	0.04279	0.02884	1.48

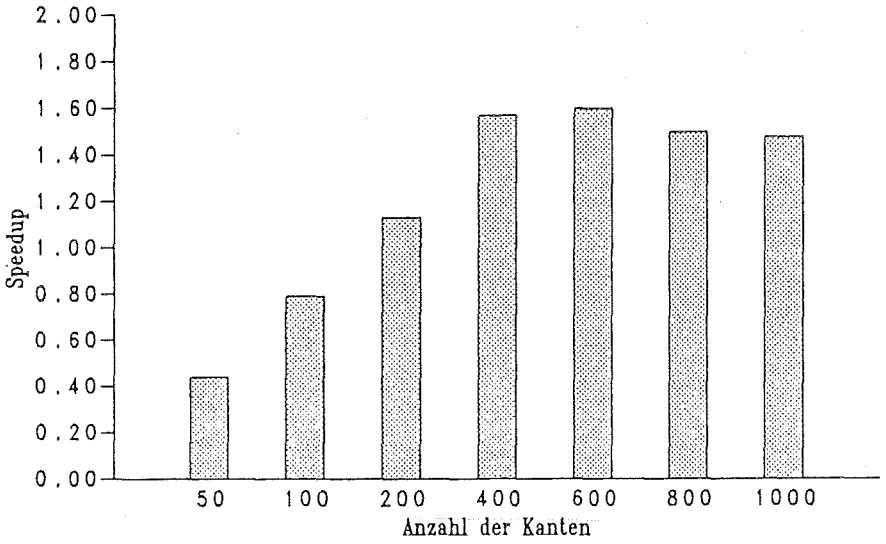


Abb. 12. Speedup-Werte erzielt durch den Algorithmus von Moore-Pape als Funktion der Kantenanzahl: 200 Knoten, Kantenbewertungen durch Zufallszahlen erzeugt

Tabelle 2. Bestimmung des kürzesten Weges von der Quelle zu allen Knoten nach dem Algorithmus von Moore-Pape; Knotenanzahl N variabel, dichtbesetzte Adjazenzmatrizen

N	ohne Multitasking	mit Multitasking	Speedup
100	0.02864	0.36369	0.08
200	0.10815	1.61063	0.07
300	0.19415	2.88946	0.07
400	0.38572	6.92313	0.06
500	0.57975	5.86263	0.10
600	0.87809	10.37201	0.08

Aus Tabelle 3 und Abb. 13 ersieht man jedoch, daß ein Speedup von fast 1.8 erreicht werden kann, wenn die Matrix dünnbesetzt ist und die Anzahl der Knoten entsprechend groß wird. Für die Adjazenzmatrizen in dieser Tabelle wurden die Kantenbewertungen sowie deren Positionen in der Matrix unabhängig durch Zufallszahlen erzeugt mit der Nebenbedingung, daß von jedem Knoten vier Kanten ausgehen, d.h. der Grad jedes Knotens vier ist. Durch die dabei gewählte Vorgehensweise kann der Grad bei einigen Knoten kleiner als vier sein.

Tabelle 3. Bestimmung des kürzesten Weges von der Quelle zu allen Knoten nach dem Algorithmus von Moore-Pape; Knotenanzahl N variabel mit konstantem Grad, dünnbesetzte Adjazenzmatrizen (ca. 4 Kanten von jedem Knoten ausgehend)

N	M	ohne Multitasking	mit Multitasking	Speedup
100	398	0.00895	0.00748	1.20
200	792	0.03489	0.02328	1.50
300	1194	0.07843	0.04797	1.63
400	1588	0.13548	0.07924	1.71
500	1968	0.23808	0.13453	1.77
600	2356	0.29392	0.16673	1.76

Der Algorithmus von Moore-Pape ist jedoch auch sehr stark abhängig von der Verteilung der Kantenbewertungen in der Adjazenzmatrix, wie man aus Tabelle 4 ersieht. In dieser Tabelle sind die Ausführungszeiten für den Algorithmus von Moore-Pape bei verschiedenen Adjazenzmatrizen mit 200 Knoten dargestellt. Bei den ersten beiden Adjazenzmatrizen ist die Anzahl der Kanten wie bei einem durchschnittlichen Straßennetz gewählt worden. Von jedem Knoten gehen etwa 4 Kanten aus und in jedem Knoten gehen etwa 4 Kanten hinein.

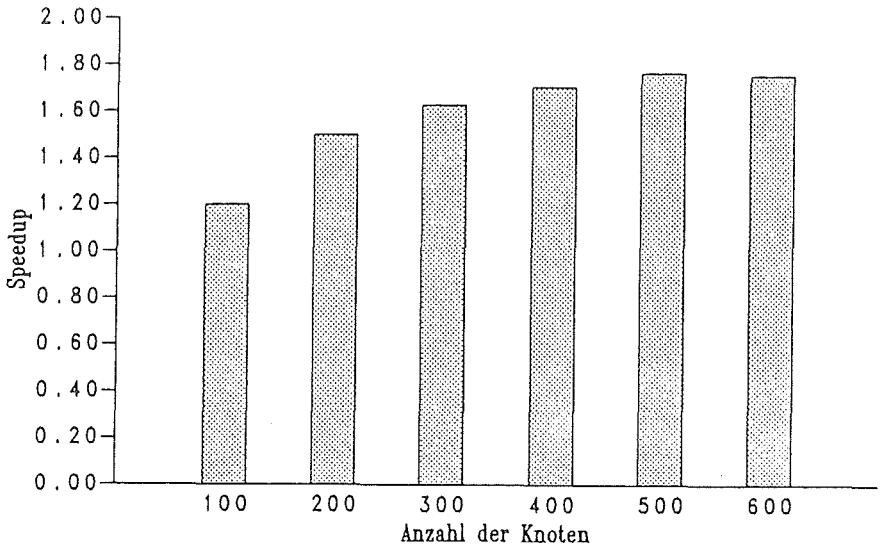


Abb. 13. Speedup-Werte erzielt durch den Algorithmus von Moore-Pape als Funktion der Knotenanzahl: dünnbesetzte Adjazenzmatrizen, Kantenbewertungen durch Zufallszahlen erzeugt, konstanter Grad

Die verschiedenen Ausführungszeiten der Programme für die ersten beiden Eingabematrizen ist durch die unterschiedliche Verteilung der Kantenbewertungen in der Matrix zu erklären. Der Speedup ist jedoch etwa gleich. Die schlechte Ausführungszeit der parallelen Version bei der Adjazenzmatrix mit nur 50 Kanten entsteht durch den zu großen Overhead. Die Arbeit, die von den beiden Tasks parallel gemacht wird, ist im Vergleich zu Kommunikations- und Synchronisationsaufwand sehr klein. Das heißt, auch bei dünnbesetzten Matrizen lohnt eine Parallelisierung wegen des zu großen Overhead nicht.

Bei den nächsten drei Adjazenzmatrizen befinden sich die Kantenbewertungen einmal im linken oberen Drittel der Matrix, einmal in der Mitte und einmal im rechten unteren Drittel der Matrix. Bei der ersten Matrix gibt es einige Wege, die von dem Quellknoten 1 ausgehen und daher ist ein Speedup zu verzeichnen. Bei den nächsten beiden Matrizen gibt es von Knoten 1 aus keine Kanten und daher sind auch keine Wege vom Knoten 1 aus zu bestimmen. Bei der parallelen Version ist daher der Overhead sehr groß. Es werden außer Knoten 1 keine weiteren Knoten in die Queue eingefügt und daher befindet sich die zweite Task nur in einer Warteschleife, während die

erste Task versucht einen Nachfolgerknoten von der Quelle zu finden, um ihn in die Queue einzufügen.
Die letzte Matrix besteht aus drei Zusammenhangskomponenten. Die Ausführungszeiten sind mit den Zeiten zu vergleichen, bei denen die Kantenbewertungen im linken oberen Drittel liegen, denn zu den anderen beiden Zusammenhangskomponenten gibt es keine Wege von Knoten 1 aus.

Tabelle 4. Bestimmung des kürzesten Weges von der Quelle zu allen Knoten nach dem Algorithmus von Moore-Pape mit unterschiedlicher Verteilung der Kantenbewertungen in der Adjazenzmatrix; Knotenanzahl N fest

N = 200	ohne Multitasking	mit Multitasking	Speedup
796 Kantenbewertungen nicht symmetrisch	0.04536	0.03054	1.49
792 Kantenbewertungen symmetrisch	0.03489	0.02328	1.50
50 Kantenbewertungen	0.00043	0.00097	0.44
275 Kantenbewertungen im linken oberen Drittel	0.01372	0.00998	1.37
279 Kantenbewertungen im mittleren Drittel	0.00014	0.00079	0.18
279 Kantenbewertungen im rechten unteren Drittel	0.00014	0.00079	0.18
784 Kantenbewertungen, 3 Zusammenhangskomponenten	0.01139	0.00846	1.35

Algorithmus von Dijkstra

Bei diesem Algorithmus wurden dieselben Testdaten verwendet wie bei dem Algorithmus von Moore-Pape.
In dem parallelen Algorithmus werden die zu untersuchenden Knoten so auf die beiden Tasks aufgeteilt, daß eine Task die erste Hälfte der Knoten und die zweite Task die zweite Hälfte bearbeitet. Verteilt man die Knoten auf die beiden Tasks so, daß jede Task jeden zweiten Knoten untersucht, so ergibt sich eine andere Reihenfolge. Dabei ist es möglich, daß sich die Ausführungszeit verbessert oder verschlechtert.
In Tabelle 5 werden die Ausführungszeiten bei einer 200 × 200 Adjazenzmatrix mit unterschiedlicher Kantenzahl dargestellt. Man findet in diesem Algorithmus keine signifikanten Unterschiede bez. der Ausführungszeiten und des Speedup gegenüber dem Algorithmus von Moore-Pape. Es zeigt sich, daß die Ausführungszeit des sequentiellen Programms ab einer bestimmten Kantenanzahl weitgehend unabhängig von der Anzahl der Kanten im Graphen ist. Graphisch wird der Speedup in Abb. 14 dargestellt.

Tabelle 5. Bestimmung des kürzesten Weges von der Quelle zu allen Knoten nach dem Algorithmus von Dijkstra; Knotenanzahl N fest, Kantenanzahl M variabel

N = 200 M	ohne Multitasking	mit Multitasking	Speedup
100	0.00157	0.00168	0.93
200	0.00329	0.00282	1.16
400	0.02489	0.01859	1.34
600	0.02712	0.02028	1.34
800	0.02787	0.02069	1.35
1000	0.02762	0.02023	1.37

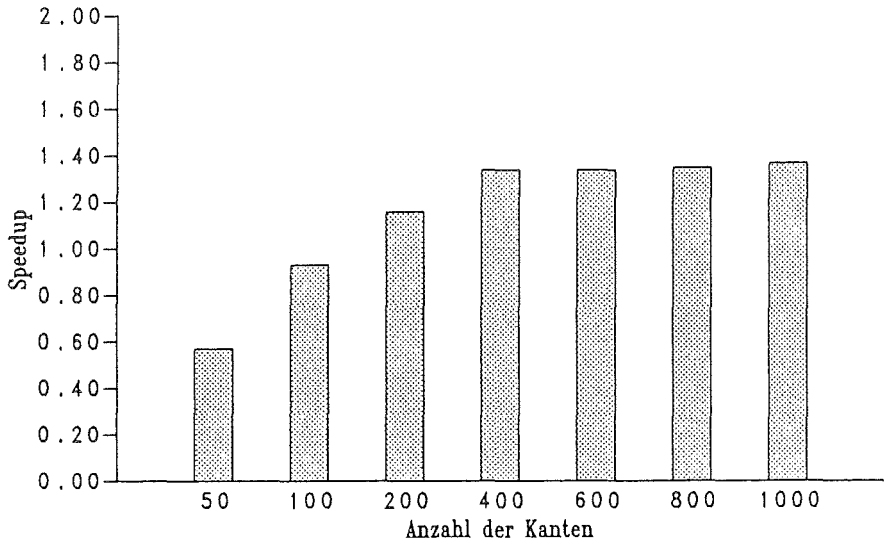


Abb. 14. Speedup-Werte erzielt durch den Algorithmus von Dijkstra als Funktion der Kantenanzahl: 200 Kanten, Kantenbewertungen durch Zufallszahlen erzeugt

Tabelle 6 und Tabelle 7 zeigen den Anstieg der Ausführungszeiten und des Speedup bei zunehmender Knotenzahl, sowohl bei einer dichtbesetzten, als auch bei einer dünnbesetzten Adjazenzmatrix. Der Speedup für die dichtbesetzten Adjazenzmatrizen ist in Abb. 15, der für die dünnbesetzten Adjazenzmatrizen mit Knoten gleichen Grades in Abb. 16 dargestellt. Aus Tabelle 6 ergibt sich insbesondere, daß auch für dichtbesetzte Matrizen eine Parallelisierung des Dijkstra-Algorithmus deutliche Gewinne bringt im Gegensatz zu dem Algorithmus von Moore-Pape.
Zusätzlich stellte sich jedoch bei den Ausführungszeiten des parallelen Programms eine starke Abhängigkeit der zahlenmäßigen Kantenbewertungen heraus. Es kann vorkommen, daß immer nur die Knoten einer Task markiert werden und dadurch beide Tasks im Verlauf des Programms immer weniger balanciert arbeiten.
Bei einem vollständigen Graphen mit 600 Knoten und Kantenbewertungen aus der Menge {1, ..., 10} ergibt sich z.B. eine parallele Ausführungszeit von 0.158 Sek. Dadurch, daß die Kantenbewertungen zwischen 1 und 10 liegen, finden beide Tasks sehr häufig das gleiche Minimum. Da die Matrix vollbesetzt ist, gibt es beispielsweise bei 600 Knoten fast von jedem Knoten aus mindestens zwei Kanten mit dem Gewicht 1. Das bedeutet, daß die Entfernung zu den von beiden Tasks vorgeschlagenen Knoten häufig gleich ist. Im Pro-

gramm wird in TASK1 abgefragt, ob das Minimum von TASK1 kleiner ist als das Minimum von TASK2. Bei Gleichheit wird der Knotenvorschlag von TASK2 gewählt. Ist nun das Minimum im Verlauf des Programms oft gleich, so sind in der Menge der Knoten, die von TASK2 untersucht wird wesentlich mehr markierte Knoten enthalten als in der Menge der Knoten, die von TASK1 untersucht wird. Dadurch ist die von TASK2 geleistete Arbeit immer geringer als die von TASK1.

Aus diesem Grund wurden bei den dichtbesetzten Matrizen Zufallszahlen zwischen 1 und 100 erzeugt. Dadurch ist eher gewährleistet, daß das absolute Minimum einmal von TASK1 und einmal von TASK2 bestimmt wird. Es kann allerdings trotzdem vorkommen, daß evtl. immer nur durch TASK1 oder TASK2 das absolute Minimum bestimmt wird. Auch das führt dann zu einer ungleichen Auslastung der Prozessoren. Im günstigsten Fall, d.h. wenn abwechselnd das globale Minimum von TASK1 und TASK2 bestimmt wird, kann der Speedup höher sein als bei den hier zufällig erzeugten Eingabedaten.

Tabelle 6. Bestimmung des kürzesten Weges von der Quelle zu allen Knoten nach dem Algorithmus von Dijkstra; Knotenanzahl N variabel, dichtbesetzte Adjazenzmatrizen

N	ohne Multitasking	mit Multitasking	Speedup
100	0.00699	0.00726	0.96
200	0.02788	0.01952	1.43
300	0.06262	0.03983	1.57
400	0.11134	0.06802	1.64
500	0.17383	0.10368	1.68
600	0.25024	0.14811	1.69

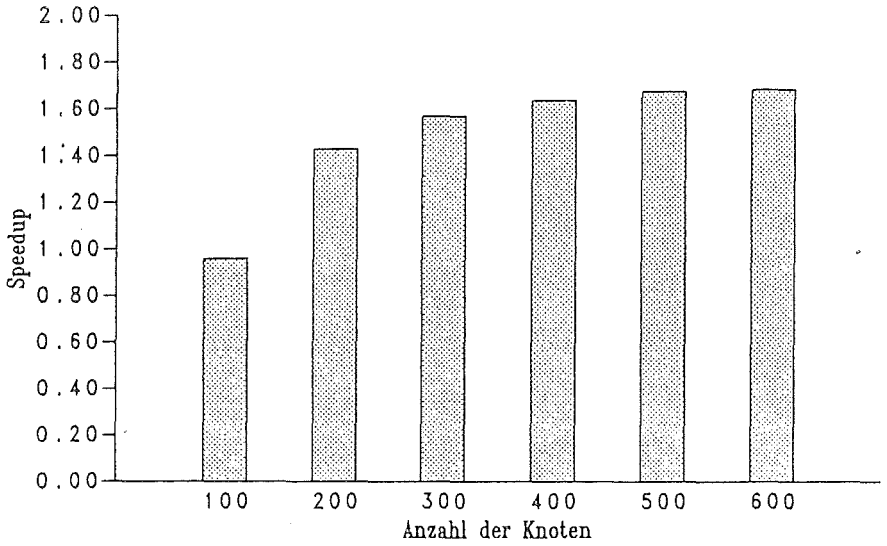


Abb. 15. Speedup-Werte erzielt durch den Algorithmus von Dijkstra als Funktion der Knotenanzahl: dichtbesetzte Adjazenzmatrizen, Kantenbewertungen durch Zufallszahlen erzeugt

Tabelle 7. Bestimmung des kürzesten Weges von der Quelle zu allen Knoten nach dem Algorithmus von Dijkstra; Knotenanzahl N variabel mit konstantem Grad, dünnbesetzte Adjazenzmatrizen (ca. 4 Kanten von jedem Knoten ausgehend)

N	M	ohne Multitasking	mit Multitasking	Speedup
100	398	0.00699	0.00668	1.05
200	792	0.02785	0.02075	1.34
300	1194	0.06264	0.04199	1.49
400	1588	0.11130	0.06671	1.67
500	1968	0.17376	0.10222	1.70
600	2356	0.25021	0.14363	1.74

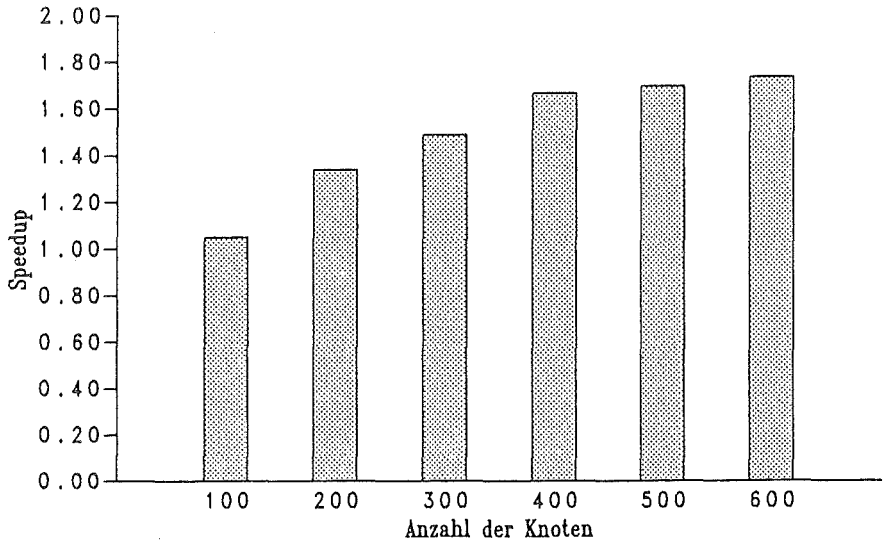


Abb. 16. Speedup-Werte erzielt durch den Algorithmus von Dijkstra als Funktion der Knotenanzahl: dünnbesetzte Adjazenzmatrizen, Kantenbewertungen durch Zufallszahlen erzeugt, konstanter Grad

Tabelle 8 zeigt die Ausführungszeiten für den Algorithmus von Dijkstra bei verschiedenen Adjazenzmatrizen mit 200 Knoten wie in Tabelle 4. Die Ausführungszeiten des Algorithmus von Dijkstra verhalten sich genauso, wie die Zeiten für den Algorithmus von Moore-Pape. Der Speedup, der bei dem Algorithmus von Moore-Pape erreicht wird, ist zwar höher als der Speedup des Algorithmus von Dijkstra, jedoch muß berücksichtigt werden, daß die Ausführungszeiten des Dijkstra-Algorithmus geringer sind als die Zeiten für den Algorithmus von Moore-Pape. Davon ausgenommen sind die Zeitmessungen, bei denen sehr wenige oder keine Wege von der Quelle aus bestimmt werden können. In diesen Fällen ist der Algorithmus von Moore-Pape etwas schneller.

Tabelle 8. Bestimmung des kürzesten Weges von der Quelle zu allen Knoten nach dem Algorithmus von Dijkstra mit unterschiedlicher Verteilung der Kantenbewertungen in der Adjazenzmatrix; Knotenanzahl N fest

N = 200	ohne Multitasking	ohne Multitasking	Speedup
796 Kantenbewertungen nicht symmetrisch	0.02786	0.01997	1.40
792 Kantenbewertungen symmetrisch	0.02785	0.02075	1.34
50 Kantenbewertungen	0.00059	0.00103	0.57
275 Kantenbewertungen im linken oberen Drittel	0.01228	0.00984	1.25
279 Kantenbewertungen im mittleren Drittel	0.00019	0.00077	0.25
279 Kantenbewertungen im rechten unteren Drittel	0.00019	0.00077	0.25
784 Kantenbewertungen, 3 Zusammenhangskomponenten	0.01165	0.00928	1.26

Algorithmus von Warshall-Floyd

Der Algorithmus von Warshall-Floyd bestimmt die kürzesten Wege zwischen allen Knotenpaaren; er benutzt Matrixmethoden und ist daher gut geeignet für dichtbesetzte Adjazenzmatrizen, wie auch die Zeitmessungen im folgenden zeigen. Es wurden wiederum dieselben Testdaten wie bei den Algorithmen von Moore-Pape und Dijkstra verwendet.
Tabelle 9 zeigt ein Ansteigen der Ausführungszeit bei zunehmender Kantenzahl im sequentiellen und im parallelen Programm. Der Speedup erreicht bei 200 Knoten und 1000 Kanten einen Wert von 1.76. Graphisch werden die Speedup-Werte in Abb. 17 dargestellt.

Tabelle 9. Bestimmung des kürzesten Weges zwischen allen Knotenpaaren nach dem Algorithmus von Warshall-Floyd; Knotenanzahl N fest, Kantenanzahl M variabel

N = 200 M	ohne Multitasking	mit Multitasking	Speedup
100	0.02894	0.02209	1.31
200	0.03080	0.02313	1.33
400	0.10368	0.06406	1.62
600	0.17623	0.10394	1.70
800	0.22319	0.12796	1.74
1000	0.23714	0.13511	1.76

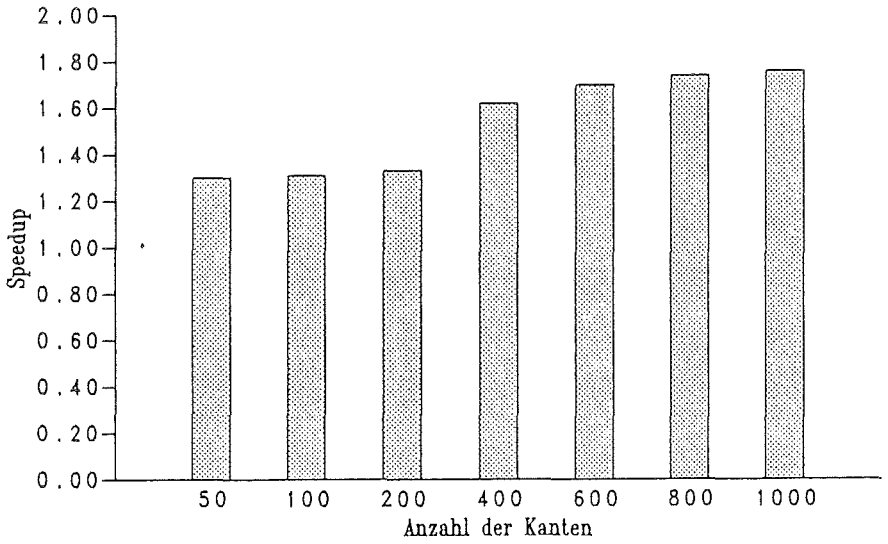


Abb. 17. Speedup-Werte erzielt durch den Algorithmus von Warshall-Floyd als Funktion der Kantenanzahl: 200 Knoten, Kantenbewertungen durch Zufallszahlen

Aus Tabelle 10 (vollbesetzte Adjazenzmatrizen) und Tabelle 11 (konstanter Grad) erkennt man, daß auch bei zunehmender Knotenzahl die Ausführungszeit sequentiell und parallel ansteigt. Der erreichte Speedup liegt hier über 1.8, wie auch in Abb. 18 und Abb. 19 dargestellt.

Zu Beginn der Implementierung wurde die Adjazenzmatrix mit N × N dimensioniert. Es zeigte sich, daß die Ausführungszeiten für 400 Knoten sowohl im sequentiellen als auch im parallelen Programm langsamer waren als für 500 oder 600 Knoten. Dies konnte auf häufige Bankkonflikte zurückgeführt werden. Dadurch, daß die Matrix in dem Algorithmus zeilenweise durchlaufen wird, wurde bei N = 400 immer auf die gleiche Bank zugegriffen. Aus diesem Grund wurde die Adjazenzmatrix mit N+1 × N dimensioniert, um die Bankkonflikte zu verringern. Daraus ergab sich für fast alle gerechneten Testdaten eine Verkürzung der Ausführungszeiten sowohl im sequentiellen als auch im parallelen Programm. Der Speedup erhöhte sich wesentlich, da diese Änderung auch eine Reduzierung der Bankkonflikte beider Tasks zur Folge hatte. Bei einem vollständigen Graphen mit 400 Knoten erhöhte sich der Speedup von 1.67 auf 1.88. Die Ausführungszeit für den sequentiellen Algorithmus reduzierte sich um den Faktor 2.5, die Rechenzeit des parallelen Programms verringerte sich um den Faktor 2.8.

Aufgrund der trotzdem noch vorhandenen Bankkonflikte ergibt sich kein ganz gleichmäßiger Anstieg der Speedup-Werte.

Tabelle 10. Bestimmung des kürzesten Weges zwischen allen Knotenpaaren nach dem Algorithmus von Warshall-Floyd; Knotenanzahl N variabel, dichtbesetzte Adjazenzmatrizen

N	ohne Multitasking	mit Multitasking	Speedup
100	0.05211	0.03205	1.63
200	0.35473	0.19836	1.79
300	1.20747	0.66723	1.81
400	2.51653	1.33598	1.88
500	5.35001	2.93375	1.82
600	8.45738	4.61515	1.83

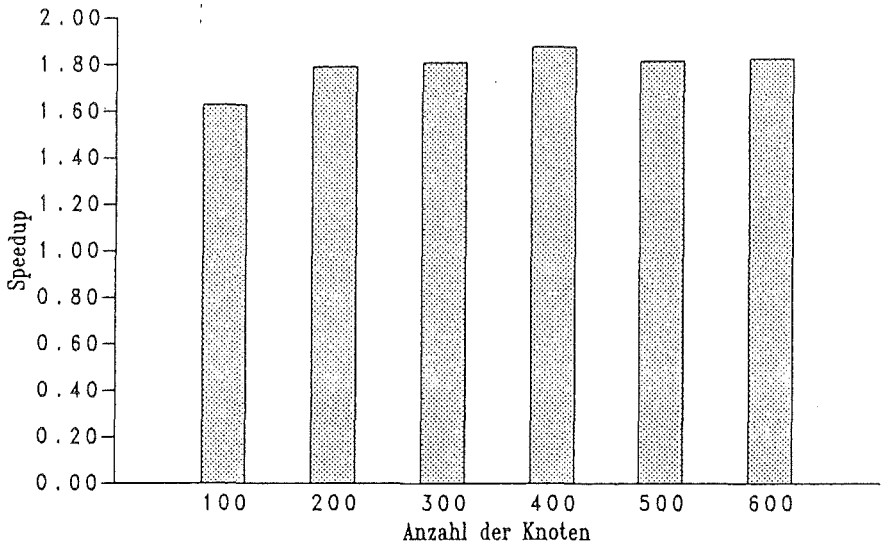


Abb. 18. Speedup-Werte erzielt durch den Algorithmus von Warshall-Floyd als Funktion der Knotenanzahl: dichtbesetzte Adjazenzmatrizen, Kantenbewertungen durch Zufallszahlen erzeugt

Tabelle 11. Bestimmung des kürzesten Weges zwischen allen Knotenpaaren nach dem Algorithmus von Warshall-Floyd; Knotenanzahl N variabel mit konstantem Grad, dünnbesetzte Adjazenzmatrizen (ca. 4 Kanten von jedem Knoten ausgehend)

N	M	ohne Multitasking	mit Multitasking	Speedup
100	398	0.03417	0.02222	1.54
200	792	0.19732	0.11370	1.74
300	1194	0.69512	0.38974	1.78
400	1588	1.47580	0.79464	1.86
500	1968	2.94138	1.63392	1.80
600	2356	4.67785	2.57571	1.82

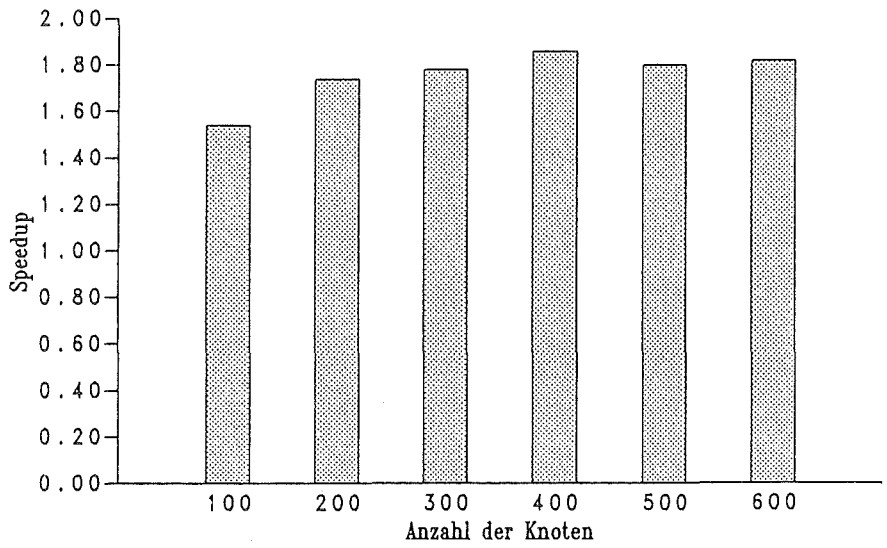


Abb. 19. Speedup-Werte erzielt durch den Algorithmus von Warshall-Floyd als Funktion der Knotenanzahl: dünnbesetzte Adjazenzmatrizen, Kantenbewertungen durch Zufallszahlen erzeugt, konstanter Grad

In Tabelle 12 sind die Ausführungszeiten für den Algorithmus von Warshall-Floyd bei verschiedenen Adjazenzmatrizen mit 200 Knoten wie in Tabelle 4 und Tabelle 8 dargestellt. Man erkennt, daß die Ausführungszeit sowohl sequentiell als auch parallel mit zunehmender Kantenzahl ansteigt und daß sich der Speedup erhöht. Wie die Kantenbewertungen in der Matrix angeordnet sind, ist bei diesem Algorithmus nicht von Bedeutung, da die kürzesten Entfernungen zwischen allen Knotenpaaren bestimmt werden sollen. Man erhält nicht so starke Speedup-Unterschiede wie bei den Algorithmen von Moore-Pape und Dijkstra. Selbst bei wenigen Kanten erhält man bei diesem Algorithmus noch einen Speedup, da die parallel auszuführende Arbeit immer noch groß genug ist.

Tabelle 12. Bestimmung des kürzesten Weges zwischen allen Knotenpaaren nach dem Algorithmus von Warshall-Floyd mit unterschiedlicher Verteilung der Kantenbewertungen in der Adjazenzmatrix; Knotenanzahl N fest

N = 200	ohne Multitasking	mit Multitasking	Speedup
796 Kantenbewertungen nicht symmetrisch	0.23031	0.13206	1.74
792 Kantenbewertungen Symmetrisch	0.19731	0.11362	1.74
50 Kantenbewertungen	0.02824	0.02178	1.30
275 Kantenbewertungen im linken oberen Drittel	0.05179	0.03452	1.50
279 Kantenbewertungen im mittleren Drittel	0.05329	0.03582	1.49
279 Kantenbewertungen im rechten unteren Drittel	0.05329	0.03521	1.51
784 Kantenbewertungen, 3 Zusammenhangskomponenten	0.09922	0.06037	1.64

5.2 MINIMALE SPANNENDE BÄUME

Von den drei vorgestellten Algorithmen zur Bestimmung des minimalen spannenden Baumes in Kapitel 4.2 wurde der Algorithmus von Prim-Dijkstra auf der CRAY X-MP/22 implementiert. Als Eingabe wurden verschiedene Adjazenzmatrizen ungerichteter zusammenhängender Graphen gewählt. Auch für diesen Algorithmus wurden die symmetrischen Matrizen durch Zufallszahlen erstellt. Die Kantenbewertungen und bei den dünnbesetzten Adjazenzmatrizen auch ihre Positionen in der Matrix, wurden unabhängig voneinander zufällig berechnet. Die Bewertungen der vollbesetzten Matrizen liegen zwischen 1 und 100, die Bewertungen der dünnbesetzten Matrizen zwischen 1 und 10.
Da der parallele Algorithmus die Knoten in einer anderen Reihenfolge bearbeitet als der sequentielle, kann der als Lösung erhaltene minimale spannende Baum andere Knoten enthalten als die Lösung des sequentiellen Algorithmus. Das Gewicht beider Bäume ist jedoch gleich.

Tabelle 13 zeigt, daß für dichtbesetzte Adjazenzmatrizen mit zunehmender Knotenzahl die Ausführungszeiten des sequentiellen und des parallelen Programms monoton ansteigen. Bei einer vollbesetzten Adjazenzmatrix erhält man somit bei 600 Knoten einen Speedup von 1.35 (siehe Abb. 21). Der Speedup ist bei diesem Algorithmus nicht so eindrucksvoll wie bei den vorherigen. Dies liegt darin begründet, daß in jeder Iteration zweimal zwischen den beiden Tasks synchronisiert werden muß. Das bewirkt einen großen Overhead.
Vergleicht man die sequentiellen Ausführungszeiten für eine dichtbesetzte Adjazenzmatrix mit den sequentiellen Ausführungszeiten einer dünnbesetzten Matrix in Tabelle 14, so kann man vermuten, daß das sequentielle Programm unabhängig von der Kantenzahl ist. Diese Vermutung ist durch verschiedene Testläufe für N = 400 bestätigt worden. Die parallelen Ausführungszeiten sind jedoch bei den dünnbesetzten Adjazenzmatrizen geringer als bei einer dichtbesetzten Matrix. Auch hier gilt wieder wie bei dem Algorithmus von Dijkstra zur Bestimmung der kürzesten Wege, daß die Ausführungszeit abhängig ist von der Auslastung der Prozessoren. Als Speedup erreicht man bei den dünnbesetzten Adjazenzmatrizen mit 600 Knoten den Wert 1.50 (siehe Abb. 20). Ein viel höherer Speedup ist wegen des zu großen Overhead nicht erreichbar.

Tabelle 13. Bestimmung eines minimalen spannenden Baumes nach dem Algorithmus von Prim-Dijkstra; Knotenanzahl N variabel, dichtbesetzte Adjazenzmatrizen

N	ohne Multitasking	mit Multitasking	Speedup
100	0.01000	0.01317	0.76
200	0.04020	0.03744	1.07
300	0.09033	0.07623	1.18
400	0.16181	0.12832	1.26
500	0.25114	0.19176	1.31
600	0.36254	0.26935	1.35

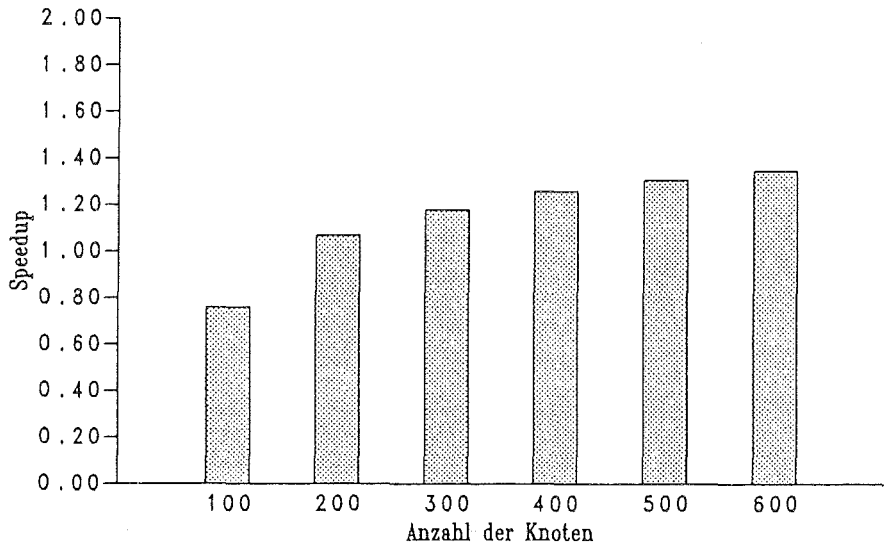


Abb. 20. Speedup-Werte erzielt durch den Algorithmus von Prim-Dijkstra als Funktion der Knotenanzahl: dünnbesetzte Adjazenzmatrizen, Kantenbewertungen durch Zufallszahlen erzeugt, konstanter Grad

Tabelle 14. Bestimmung eines minimalen spannenden Baumes nach dem Algorithmus von Prim-Dijkstra; Knotenanzahl N variabel mit konstantem Grad, dünnbesetzte Adjazenzmatrizen
(ca. 4 Kanten von jedem Knoten ausgehend)

N	M	ohne Multitasking	mit Multitasking	Speedup
100	398	0.01000	0.01302	0.77
200	792	0.04019	0.03655	1.10
300	1194	0.09038	0.07106	1.27
400	1588	0.16188	0.11679	1.39
500	1968	0.25114	0.17341	1.45
600	2356	0.36246	0.24157	1.50

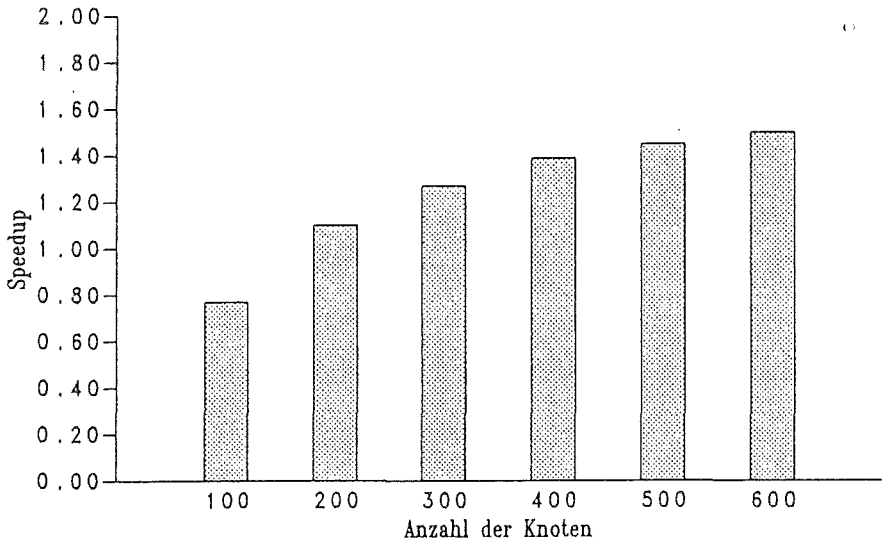


Abb. 21. Speedup-Werte erzielt durch den Algorithmus von Prim-Dijkstra als Funktion der Knotenanzahl: dichtbesetzte Adjazenzmatrizen, Kantenbewertungen durch Zufallszahlen erzeugt

5.3 TRAVELING SALESMAN PROBLEM

Von den Algorithmen zur Lösung des Traveling Salesman Problems ist der Unterteilungsalgorithmus von Karp auf der CRAY X-MP/22 implementiert worden. Als Eingabe erhält der Algorithmus die beiden Felder XLOC und YLOC, die die x- und y-Koordinaten enthalten. Diese Werte sind durch Zufallszahlen zwischen 1 und 10 generiert worden. Weiterhin werden die Felder X und Y eingelesen, die nach x- bzw. y-Koordinaten sortiert sind. Das heißt, in der ersten Komponente von Array X steht der Index der kleinsten x-Koordinate aus Array XLOC. Außerdem werden die Felder XASSOC und YASSOC eingelesen, die die Positionen der Elemente aus X im Array Y angeben, bzw. die Positionen der Elemente von Y im Array X.
Der rekursive Algorithmus von Karp wurde durch eine iterative Version ersetzt, wobei die Rekursion durch einen Stack simuliert wurde.
Bei der Darstellung des Algorithmus in Kapitel 4.3.3 wird die Anzahl der Tasks von den zur Verfügung stehenden Prozessoren sowie der Partitions-

nummer bestimmt (Anzahl der Tasks = max {P div 2^{i-1},1}). Alle erzeugten Tasks können dann lediglich das Kopieren der Elemente in die neuen Teilfelder parallel ausführen. Für die Implementierung auf einem Rechner mit zwei Prozessoren würde das bedeuten, daß nur für die erste Unterteilung der Felder eine zweite Task gebildet wird, so daß das Kopieren parallel ausgeführt werden kann. Der ganze weitere Rechenaufwand muß dann sequentiell ausgeführt werden. Aus diesem Grund ist der Unterteilungsalgorithmus von Karp für die Implementierung auf der CRAY X-MP/22 so modifiziert worden, daß bis auf das erste Unterteilen der Felder und das Kombinieren der letzten beiden Teilfelder der Algorithmus ohne Synchronisation parallel ausgeführt werden kann. Nachdem die erste Unterteilung beendet ist, arbeiten beide Tasks asynchron die beiden Teilfelder ab, bis beide eine Teiltour als Lösung erhalten. Nun muß eine Task diese beiden Teiltouren zu einer Näherungslösung des Algorithmus kombinieren. Durch diese Art der Parallelisierung ist das Unterprogramm *count_b_indices* nicht mehr nötig.

Da alle Unterprogramme in dem Programm von beiden Tasks aufgerufen werden, muß der Code reentrant sein. Damit diese Voraussetzung gewährleistet ist, muß als Compiler-Option ALLOC=STACK angegeben werden.

Dadurch, daß das Programm sehr viele Variablen benötigt, insbesondere die sechs Felder für die x- und y-Koordinaten, die Felder sortiert nach x- und y-Koordinaten, sowie die Felder für die Positionen der Knoten in diesen beiden Feldern, ist die Ausführungszeit für das parallele Programm äußerst langsam. Beispielsweise wurde für 400 Knoten ein Speedup von 1.38 erreicht. Für 500 und 600 Knoten betrug die Beschleunigung nur noch 1.33. Durch Zeitmessungen in den verschiedenen Unterprogrammen, konnte festgestellt werden, daß die Ausführungszeit der Unterprogramme im sequentiellen Programm für die gleichen Partitionen geringer war als für die Unterprogramme im parallelen Programm. Deshalb wurde untersucht, ob sich die parallelen Ausführungszeiten unterscheiden, je nachdem, ob die benötigten Variablen in TASK COMMON-Blöcken vereinbart oder als Parameter übergeben werden. Dabei stellte sich heraus, daß die Parameterübergabe bei größeren Feldern in dem parallelen Programm mehr Zeit kostete als im sequentiellen Programm. Bei Vereinbarung der Variablen im TASK COMMON-Bereich dauerte ein Unterprogrammaufruf ebenso lange wie in dem entsprechenden sequentiellen Programm bei Vereinbarung der Variablen in COMMON-Blöcken. Trotzdem erreichte der Speedup nicht den erwarteten Wert. Aus diesem Grund wurde das parallele Programm so geändert, daß es ohne die Compiler-Option

ALLOC=STACK übersetzt werden konnte. Dazu mußten alle Unterprogramme, die von beiden Tasks aufgerufen werden, zweimal vorhanden sein. Dadurch wurde die Ausführungszeit z.B. bei N = 600 um den Faktor 1.4 kürzer. Tabelle 15 zeigt die Ausführungszeiten für das sequentielle und das parallele Programm ohne die Compiler-Option ALLOC=STACK. Graphisch ist der Speedup in Abb. 22 dargestellt.

Tabelle 15. Lösung des euklidischen Traveling Salesman Problems nach dem Unterteilungsalgorithmus von Karp; Knotenanzahl N variabel

N	ohne Multitasking	mit Multitasking	Speedup
50	0.00247	0.00203	1.22
100	0.00518	0.00351	1.48
200	0.01074	0.00676	1.59
300	0.01547	0.00941	1.64
400	0.02237	0.01311	1.71
500	0.02493	0.01464	1.71
600	0.03202	0.01848	1.73

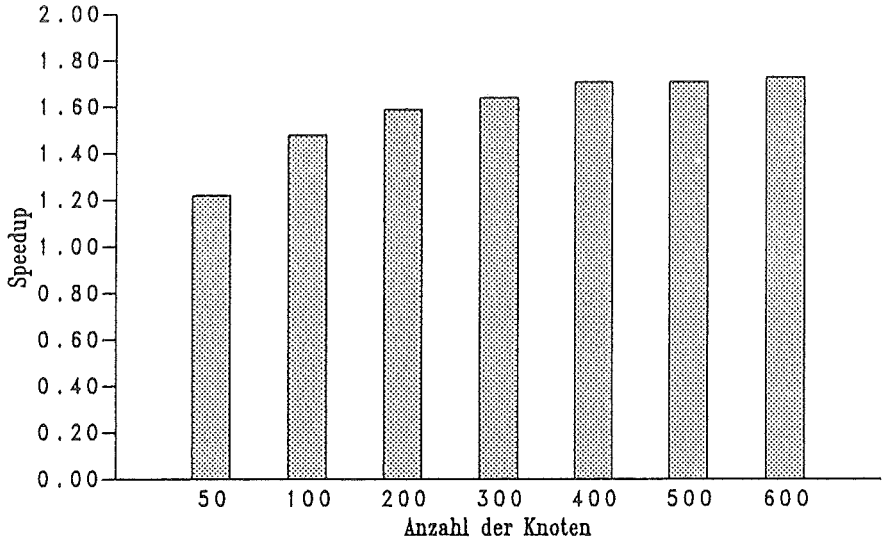


Abb. 22. Speedup-Werte erzielt durch den Unterteilungsalgorithmus von Karp: x- und y-Koordinaten durch Zufallszahlen erzeugt

6.0 SCHLUßBEMERKUNGEN

Der erreichte Speedup der implementierten Graphenalgorithmen ist nicht so hoch wie bei Algorithmen der Linearen Algebra. Dies liegt daran, daß bei den hier implementierten Algorithmen eine häufige Synchronisation und Kommunikation zwischen den beiden Tasks erforderlich ist. Bezeichnenderweise ist der Speedup bei dem Algorithmus von Warshall-Floyd, der auf Matrixmethoden basiert, am höchsten.
Im Verlauf der Implementierung ergab sich eine Reihe von Effekten (Bankkonflikte, Compiler-Optionen, ungleiche Auslastung der Prozessoren, geringe Granularität), die den Speedup deutlich beeinflussen können. Es ist daher anzunehmen, daß mit weiterer Feinarbeit bei der Implementierung noch Verbesserungen der parallelen Performance möglich sind.
Da bisher praktisch noch kaum Implementationen von Graphenalgorithmen auf MIMD-Rechnern durchgeführt wurden, konnte hier nicht auf Erfahrungen und bekannte Techniken zurückgegriffen werden. Dies ist bei Algorithmen der Linearen Algebra in viel größerem Umfang möglich.
Möglicherweise kann der Speedup noch erhöht werden, wenn die Vektorisierung, die nicht Ziel der Arbeit war, noch stärker ausgenutzt wird.

Die in Kapitel 4 vorgestellten Algorithmen sind unter dem Gesichtspunkt dargestellt, daß sie auf mehr als zwei Prozessoren implementiert werden können. Einige der auf der CRAY X-MP/22 implementierten Algorithmen sind daher so modifiziert worden, daß die beiden zur Verfügung stehenden Prozessoren so gut wie möglich ausgenutzt werden.
Von Interesse ist jedoch auch, wie sich die Algorithmen verhalten, wenn mehr als zwei Prozessoren vorhanden sind. Bei dem Algorithmus von Moore-Pape zur Bestimmung kürzester Wege, kann erwartet werden, daß der Speedup für beliebig viele Prozessoren eine konstante obere Schranke hat. Die Ursache dafür liegt in dem exklusiven Zugriff aller Prozesse auf die Queue. In [Quinn,83] wurde daher der Versuch gemacht, den sequentiellen Zugriff auf die Queue zu umgehen, indem jedem Prozeß eine eigene Queue zur Verfügung steht und nur Elemente aus der eigenen Queue entfernt werden können. Dies kann ohne Synchronisation geschehen. Will ein Prozeß einen Knoten in eine Queue einfügen, so hängt die Queue, in die der Knoten eingefügt wird,

von der Knotennummer ab. Eine Strategie wäre z.B. Knoten i in die Queue mit der Nummer (i mod P) + 1 einzufügen.

Die Effizienz des Algorithmus von Warshall-Floyd ist offensichtlich hoch, wenn die Anzahl der Knoten N ein Vielfaches der Anzahl der Prozessoren ist. Dann ist die zu leistende Arbeit gleichmäßig auf alle Prozessoren verteilt. Ist P kein Teiler von N, so müssen die Prozesse, die zuerst fertig werden, auf alle anderen warten, bevor sie mit der nächsten Iteration beginnen können.

Insgesamt kann festgestellt werden, daß fast alle implementierten Algorithmen für MIMD-Rechner geeignet sind, zumindest bei nicht zu kleinen Problemen. Lediglich der Algorithmus von Moore-Pape ist bei dichtbesetzter Adjazenzmatrix auf MIMD-Rechnern völlig unbrauchbar.

[AHU,74] Aho, A. V., Hopcroft, J. E. und Ullman, J. D.
The Design and Analysis of Computer Algorithms
Addison-Wesley Publishing Company, 1974

[CRAY,222] CRAY Computer Systems: Technical Note
Multitasking User Guide
Revision A
CRAY-Research Inc., SN-0222, January 1985

[DeYo,81] Deo, N. und Yoo, Y. B.
Parallel algorithms for the minimum spanning tree problem
Proceedings of the 1981 International Conference on Parallel Processing, IEEE, New York, pp. 188-189, 1981

[Dijk,59] Dijkstra, E.
A note on two problems in connexion with graphs
Numer. Math. 1, pp. 269-271, 1959

[DöMü,73] Dörfler, W. und Mühlbacher, J.
Graphentheorie für Informatiker
Sammlung Göschen, Band 6016, de Gruyter, 1973

[DPL,80] Deo, N., Pang, C. Y. und Lord, R. E.
Two parallel algorithms for shortest path problems
Proceedings of the 1980 International Conference on Parallel Processing, IEEE, New York, pp. 244-253, 1980

[Floy,62] Floyd, R. W.
Algorithm 97: shortest path
Communications of the ACM 5, 6, p. 345, 1962

[Fly,66] Flynn, M. J.
Very high-speed computing systems
Proc. of the IEEE 54, pp. 1901-1909, 1966

[Gil,81] Giloi, W. K.
Rechnerarchitektur
Springer-Verlag, 1981

[Hoß,81] Hoßfeld, F.
Parallele Algorithmen
KFA Jülich, Jül-Spez-125, 1981

[HwBr,84] Hwang, K. und Briggs, F. A.
Computer Architecture and parallel processing
McGraw-Hill Book Company, 1984

[KFA52,85] *Einführung in die Benutzung der CRAY*
Informationen für die Benutzer der zentralen Rechenanlagen
Kernforschungsanlage Jülich, 1985
KFA-ZAM-0052-CRAY

[Krus,56] Kruskal, J. B.
On the shortest subtree of a graph and the traveling salesman problem
Proc. Amer. Math. Soc. 7 (Feb.), pp. 48-50, 1956

[Lars,84] Larson, J. L.
Multitasking on the CRAY X-MP-2 Multiprocessor
IEEE Computer, Vol. 17, No. 7, pp. 62-69, 1984

[LMSK,63] Little, J. D. C., Murty, K. G., Sweeney, D. W. und Karel, C.
An algorithm for the traveling salesman problem
Operations Research 11 (6), pp. 972-989, 1963

[MaDe,82] Mateti, P. und Deo, N.
Parallel algorithms for the single source shortest path problem
Computing, 29, pp. 31-49, 1982

[Möh,84] Möhring, R.
Graphentheorie
Vorlesungsmanuskript SS1984

[Moh,82] Mohan, J.
A study in parallel computation - the traveling salesman problem
Technical Report CMU-CS-82-136, Computer Science Department, Carnegie-Mellon University, 1982

[Moh,83] Mohan, J.
Experience with two parallel programs solving the traveling salesman problem
Proceedings of the 1983 International Conference on Parallel Processing, IEEE, New York, pp. 191-193, 1983

[Moore,57] Moore, E. G.
The shortest path through a maze
Proc. Internat. Symp. on Theory of Switching, pp. 285-292, 1957

[Pape,80] Pape, U.
Algorithm 562: shortest path lengths
ACM Trans. on Math. Software 6, pp. 450-455, 1980

[Prim,57] Prim, R. C.
Shortest connection networks and some generalizations
Bell Syst. Tech. J. 36, pp. 1389-1401, 1957

[QuDe,84] Quinn, M. J. und Deo, N.
Parallel Graph Algorithms
Computing Surveys, Vol. 16, No. 3, pp. 319-348, 1984

[Quinn,83] Quinn, M. J.
The design and analysis of algorithms and data structures for the efficient solution of graph theoretic problems on MIMD computers
Ph.D. dissertation, Computer Science Dept., Washington State Univ., Pullman, Wash., 1983

[RND,77] Reingold, E., Nievergelt, J. und Deo, N.
Combinatorial Algorithms: Theory and Practice
Prentice-Hall, Englewood Cliffs, New Jersey, 1977

[SDK,83] Syslo, M. M., Deo, N. und Kowalik, J. S.
Discrete Optimization Algorithms with PASCAL programs
Prentice-Hall, Englewood Cliffs, New Jersey, 1983

[Sol,77] Sollin, M.
An algorithms attributed to Sollin
In: Introduction to the Design and Analysis of Algorithms
S. E. Goodman, S. T. Hedetniemi (Hrsg.)
McGraw-Hill, New York, Kap. 5.5, 1977

[Sto,75] Stone, H. S. (Hrsg.)
Introduction to Computer Architecture
SRA, Chicago, 1975

[Yoo,83] Yoo, Y. B.
Parallel Processing for some network optimization problems
Ph.D. dissertation, Computer Science Dept.,
Washington State Univ., Pullman, Wash., 1983